

TILED ALGORITHMS FOR MATRIX COMPUTATIONS ON MULTICORE
ARCHITECTURES

by

Henricus M Bouwmeester

B.S., Colorado Mesa University, 1998

M.S., Colorado State University, 2000

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Applied Mathematics

2012

This thesis for the Doctor of Philosophy degree by
Henricus M Bouwmeester
has been approved
by

Julien Langou, Advisor
Lynn Bennethum, Chair
Gita Alaghband
Elizabeth R. Jessup
Stephen Billups

October 26, 2012

Bouwmeester, Henricus M (Ph.D., Applied Mathematics)

Tiled Algorithms for Matrix Computations on Multicore Architectures

Thesis directed by Associate Professor Julien Langou

ABSTRACT

Current computer architecture has moved towards the multi/many-core structure. However, the algorithms in the current sequential dense numerical linear algebra libraries (*e.g.* LAPACK) do not parallelize well on multi/many-core architectures. A new family of algorithms, the *tile algorithms*, has recently been introduced to circumvent this problem. Previous research has shown that it is possible to write efficient and scalable tile algorithms for performing a Cholesky factorization, a (pseudo) LU factorization, and a QR factorization. The goal of this thesis is to study tiled algorithms in a multi/many-core setting and to provide new algorithms that exploit the current architecture to improve performance relative to current state-of-the-art libraries while maintaining the stability and robustness of these libraries.

In Chapter 2, we confront the problem of computing the inverse of a symmetric positive definite matrix with tiled algorithms. We observe that, using a dynamic task scheduler, it is relatively painless to translate existing LAPACK code to obtain a ready-to-be-executed tile algorithm. However we demonstrate that nontrivial compiler techniques (array renaming, loop reversal and pipelining) need to be applied to further increase the parallelism of our application, both theoretically and experimentally.

Chapter 3 revisits existing algorithms for the QR factorization of rectangular matrices composed of $p \times q$ tiles, where $p \geq q$, for an unlimited number of processors. Within this framework, we study the critical paths and performance of algorithms such as SAMEH-KUCK, FIBONACCI, GREEDY, and those found within PLASMA. We

also provide a monotonically increasing function to transform the elimination list of a coarse-grain algorithm to a tiled algorithm. Although the optimality from the coarse-grain GREEDY algorithm does not translate to the tiled algorithms, we propose a new algorithm and show that it is optimal in the tiled context.

In Chapters 2 and 3, our context includes an unbounded number of processors. The exercise was to find algorithmic variants with short critical paths. Since the number of resources is unlimited, any task is executed as soon as all its dependencies are satisfied. In Chapters 4 and 5, we set ourselves in the more realistic context of bounded number of processors. In this context, at a given time, the number of ready-to-go tasks can exceed the number of available resources, and therefore a schedule which prescribes which tasks to execute when needs to be defined. For the Cholesky factorization, we study standard schedules and find that the critical path schedule is best. We also derive a lower bound on the time to solution of the optimal schedule. We conclude that the critical path schedule is nearly optimal for our study. For the QR factorization problem, we study the problem of optimizing the reduction trees (therefore the algorithm) and the schedule simultaneously. This is considerably harder than the Cholesky factorization where the algorithm is fixed and so, for Cholesky factorization, we are concerned only with schedules. We provide a lower bound for the time to solution for any tiled QR algorithm and any schedule. We also show that, in some cases, the optimal algorithm for an unbounded number of processors (found in Chapter 3) cannot be scheduled to solve optimally the combined problem. We compare our algorithms and schedules with our lower bound.

Finally, in Chapter 6 we study a recursive tiled algorithm in the context of matrix multiplication using the Winograd-Strassen algorithm using a dynamic task scheduler. Whereas most implementations obtain either one or two levels of recursion, our implementation supports any level of recursion.

The form and content of this abstract are approved. I recommend its publication.

Approved: Julien Langou

TABLE OF CONTENTS

Figures	viii
Tables	xi
<u>Chapter</u>	
1. Introduction	1
2. Cholesky Inversion	8
2.1 Tile in-place matrix inversion	10
2.2 Algorithmic study	13
2.3 Conclusion and future work	15
3. QR Factorization	17
3.1 The QR factorization algorithm	20
3.1.1 Kernels	23
3.1.2 Elimination lists	26
3.1.3 Execution schemes	28
3.2 Critical paths	30
3.2.1 Coarse-grain algorithms	30
3.2.1.1 SAMEH-KUCK algorithm	30
3.2.1.2 FIBONACCI algorithm	31
3.2.1.3 GREEDY algorithm	31
3.2.2 Tiled algorithms	32
3.3 Experimental results	51
3.4 Conclusion	60
4. Scheduling of Cholesky Factorization	67
4.1 ALAP Derived Performance Bound	68
4.2 Critical Path Scheduling	69
4.3 Scheduling with synchronizations	71
4.4 Theoretical Results	73
	vi

4.5	Toward an α_{opt}	75
4.6	Related Work	75
4.7	Conclusion and future work	76
5.	Scheduling of QR Factorization	78
5.1	Performance Bounds	78
5.2	Optimality	80
5.3	Elimination Tree Scheduling	81
5.4	Conclusion	83
6.	Strassen Matrix-Matrix Multiplication	84
6.1	Strassen-Winograd Algorithm	85
6.2	Tiled Strassen-Winograd Algorithm	86
6.3	Related Work	91
6.4	Experimental results	93
6.5	Conclusion	96
7.	Conclusion	97
<u>Appendix</u>		
A.	Integer Programming Formulation of Tiled QR	99
A.1	IP Formulation	99
A.1.1	Variables	99
A.1.2	Constraints	100
A.1.2.1	Precedence constraints	105
A.1.3	Objective function	107
<u>References</u>		108

FIGURES

Figure

1.1	Three variants for the Cholesky decomposition	3
1.2	Data layout of tiled matrix	5
1.3	Three variants of the Cholesky decomposition applied to a tiled matrix of 4 × 4 tiles.	5
2.1	DAGs of Step 3 of the Tile Cholesky Inversion ($t = 4$).	12
2.2	Scalability of Algorithm 2.1 (in place) and its out-of-place variant intro- duced in § 2.2, using our dynamic scheduler against vecLib, ScaLAPACK and LAPACK libraries.	13
2.3	Impact of loop reversal on performance.	15
3.1	Icon representations of the kernels	42
3.2	Critical Path length for the weighted FLATTREE on a matrix of 4 × 4 tiles.	43
3.3	Illustration of first and second parts of the proof of Theorem 3.14 using the FIBONACCI algorithm on a matrix of 15 × 2 tiles.	49
3.4	GREEDY versus GRASAP on matrix of 15 × 2 tiles.	50
3.5	Tiled matrices of $p \times q$ where the critical path length of GRASAP is shorter than that of GREEDY for $1 \leq p \leq 100$ and $1 \leq q \leq p$	50
3.6	Upper bound and experimental performance of QR factorization - TT kernels	52
3.7	Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)	53
3.8	Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)	55
3.9	Kernel performance for double complex precision	56
3.10	Kernel performance for double precision	57
3.11	Upper bound and experimental performance of QR factorization - All kernels	58

3.12	Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)	59
3.13	Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)	60
4.1	ALAP execution for 5×5 tiles	69
4.2	Example derivation of task priorities via the Backflow algorithm	71
4.3	Theoretical results for matrix of 40×40 tiles.	74
4.4	Values of α for matrices of $t \times t$ tiles where $3 \leq t \leq 40$	75
4.5	Asymptotic efficiency versus $\alpha = p/n$ for LU decomposition and versus $\alpha = p/t^2$ for Tiled Cholesky factorization.	76
5.1	Scheduling comparisons for each of the algorithms versus the ALAP De- rived bounds on a matrix of 20×6 tiles.	79
5.2	Tail-end execution using ALAP on unbounded resources for GRASAP and FIBONACCI on a matrix of 15×4 tiles.	80
5.3	ALAP Derived bound comparison for all algorithms for a matrix of 15×4 tiles.	81
5.4	Comparison of speedup for CP Method on GRASAP, ALAP Derived bound from GRASAP, and optimal schedules for a matrix of 5×5 tiles on 1 to 14 processors	82
6.1	Task graph for the Strassen-Winograd Algorithm. Execution time pro- gresses from left to right. Large ovals depict multiplication and small ovals addition/subtraction.	86
6.2	Strassen-Winograd DAG for matrix of 4×4 tiles with one recursion. Exe- cution time progresses from left to right. Large ovals depict multiplication and small ovals addition/subtraction.	88
6.3	Required extra memory allocation for temporary storage for varying re- cursion levels.	93

6.4	Comparison of tuning parameters n_b and r	94
6.5	Scalability and Efficiency comparisons on 48 threads with matrices of $64 \times$ 64 tiles and $n_b = 200$	95
6.6	Scalabilty and efficiency comparisons executed on 12 threads with matrices of 64×64 tiles and $n_b = 200$	96

TABLES

Table

2.1	Length of the critical path as a function of the number of tiles t	13
3.1	Kernels for tiled QR. The unit of time is $\frac{n_b^3}{3}$, where n_b is the blocksize. . .	23
3.2	Time-steps for coarse-grain algorithms.	32
3.3	Time-steps for tiled algorithms.	63
3.4	Neither GREEDY nor ASAP are optimal.	63
3.5	Three schemes applied to a column whose update kernel weight is not an integer multiple of the elimination kernel weight.	64
3.6	Greedy versus PT_TT and Fibonacci (Theoretical)	65
3.7	Greedy versus PT_TT (Experimental Double)	66
3.8	Greedy versus PT_TT (Experimental Double Complex)	66
3.9	Greedy versus Fibonacci (Experimental Double)	66
3.10	Greedy versus Fibonacci (Experimental Double Complex)	66
4.1	Task Weights	67
4.2	Upper bound on speedup and efficiency for 5×5 tiles	69
5.1	Schedule lengths for matrix of 5×5 tiles	82
6.1	Strassen-Winograd Algorithm	85
6.2	Recursion levels which minimize the number of tasks for a tiled matrix of size $p \times p$	89
6.3	128×128 tiles of size $n_b = 200$	90
6.4	Comparison of the total number of tasks and critical path length for matrix of $p \times p$ tiles.	90

1. Introduction

The High-Performance Computing (HPC) landscape is trending more towards a multi/many-core architecture [8] as is evidenced by the recent projects of major chip manufacturers and reports of surveys conducted by consulting companies [52]. The computational algorithms for dense linear algebra need to be re-examined to make better use of these architectures and provide higher levels of efficiency. Some of these algorithms may have a straight forward translation from the current state-of-the-art libraries while others require much more thought and effort to gain performance increases. In this thesis, we endeavor to achieve algorithms that exploit the architecture to improve performance relative to current state-of-the-art computational libraries while maintaining the stability and robustness of these libraries.

Our research will make use of the BLAS (Basic Linear Algebra Subprograms) [15] and the LAPACK (Linear Algebra PACKage) [7] libraries. The BLAS are a standard to perform basic linear algebra operations involving vectors and matrices while LAPACK performs the more complicated and higher level linear algebra operations.

The BLAS divide numerical linear algebra operations into three distinct groupings based upon the amount of input data and the computational cost. Operations involving only vectors are considered Level 1; those involving both vectors and matrices are Level 2; and those involving only matrices are Level 3. For matrices of size $n \times n$, the Level 3 operations are most coveted since they use $O(n^2)$ data for $O(n^3)$ computations and inherently reduce the amount of memory traffic. Since the BLAS provide fundamental linear algebra operations, hardware and software vendors such as Intel, AMD, and IBM provide optimized BLAS libraries for a variety of architectures. The BLAS library can be multithreaded to make use of multi/many-core architectures and most of the vendor supplied libraries are multithreaded.

The LAPACK library is a collection of subroutines for solving most of the common problems in numerical linear algebra and was developed to make use of Level 3 BLAS

operations as much as possible. Algorithms within LAPACK are written to make use of panels which can be either a block of columns or a block of rows so that updates are performed using matrix multiplications instead of vector operations. LAPACK can make use of a multithreaded BLAS to exploit multi/many-core architectures but this may not be enough to fully exploit the capability of the architecture.

As an example, let us consider the Cholesky decomposition to factorize a symmetric positive definite (SPD) matrix into its triangular factor. There are three variants for performing the Cholesky decomposition: bordered, left-looking, and right-looking. All three work on either the upper or lower triangular portion of the matrix and produce the same triangular factor, but depending on the usage, one may have an advantage over the others. In Figure 1.1 we depict the steps involved in each variant using the lower triangular formulations. At the start of each variant, the matrix is subdivided into blocks of rows and columns, or panels, which will take advantage of the Level 3 BLAS.

The ‘bordered’ variant, as depicted in Figure 1.1a, involves a loop over three steps. The first step updates the purple row block using the already factorized green portion, the second step updates the next triangular block to be factorized (in red), and the third step performs the factorization of the triangular block. This is then repeated until the entire matrix is factorized. Note that the lower portion of the matrix is not touched by the preceding steps.

The ‘left-looking’ variant (see Figure 1.1b) involves four steps. The first step updates the triangular block in red which is then factorized in the second step, the third step updates the block column (in cyan) below the triangular block using the previous columns, and the last step updates the column block using the factorization of step 2. It is called left-looking since the algorithm does not affect the portion to the right of the current block of the matrix and only ‘looks’ to the left for its updates. The top most triangular portion of the matrix is in its final form and will not change

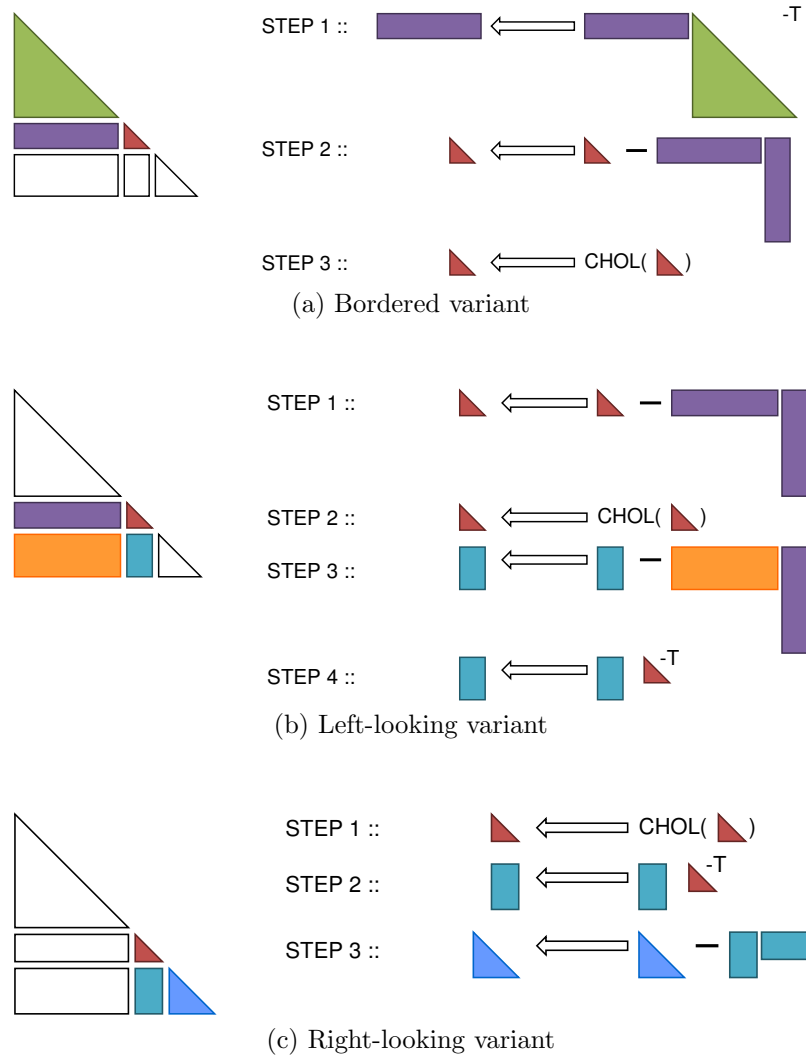


Figure 1.1: Three variants for the Cholesky decomposition

in the succeeding steps of the algorithm.

The ‘right-looking’ variant (see Figure 1.1c) involves three steps. It performs the factorization of the red triangular block, updates the block column (in *cyna*) and then updates the blue trailing matrix on the right. This is called right-looking since it does not require anything from the previous factorized matrix and pushes its updates to the right part of the matrix. The entire matrix to the left of the column block in which the algorithm is currently working is in its final form.

The advantage of the bordered variant is that it does the least amount of operations to determine if a matrix is SPD. The advantage of the right-looking variant is that it provides the most parallelism. A major disadvantage of the left-looking variant is the added fork-join that it must perform between the steps as compared to the other two variants which will negatively affect its parallel performance.

The LAPACK scheme of using panels has three distinct disadvantages which limit its performance. The first can be seen in the third step of the right-looking Cholesky decomposition (Figure 1.1c) where potentially a large symmetric rank k operation is performed; the memory architecture will bound the performance of the algorithm. Secondly, there is some impact of the synchronizations that must be performed between each step. Third, the idea of panels does not allow for fine-grained tasks. We alleviate the last two of these restrictions through the use of tiled algorithms whereas the first is overcome through the use of Level 3 BLAS operations within the tiled algorithm.

We approach this via tiling a matrix which means reordering the data of the matrix into smaller regions of contiguous memory as depicted in Figure 1.2. By varying the tile size, this data layout allows us to tune the algorithm such that the data needed for the kernels is present in the cache of the processor core. Moreover, we are able to increase the amount of parallelism and minimize the synchronizations.

Let us revisit the Cholesky decomposition as described earlier and apply each

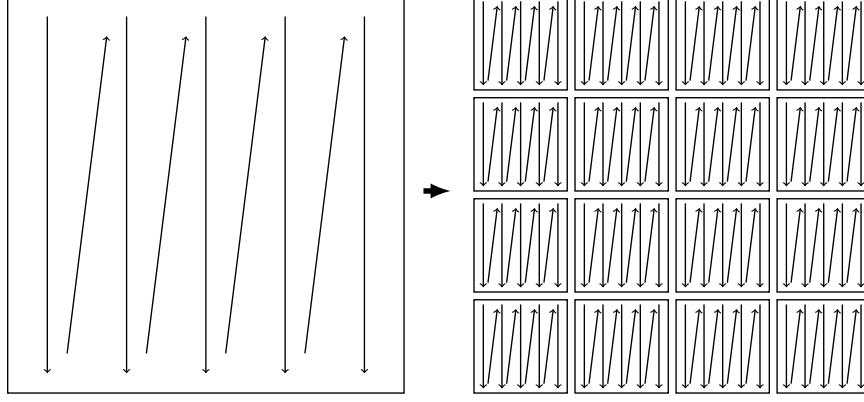


Figure 1.2: Data layout of tiled matrix

step to the tiled matrix. In Figure 1.3 we present the directed acyclic graphs (DAG) for the three variants on a tiled matrix of 4×4 tiles. In each of the DAGs, the tasks are represented as the nodes and the data dependencies are the edges. The dashed horizontal lines designate a full sweep through all of the steps in each algorithm.

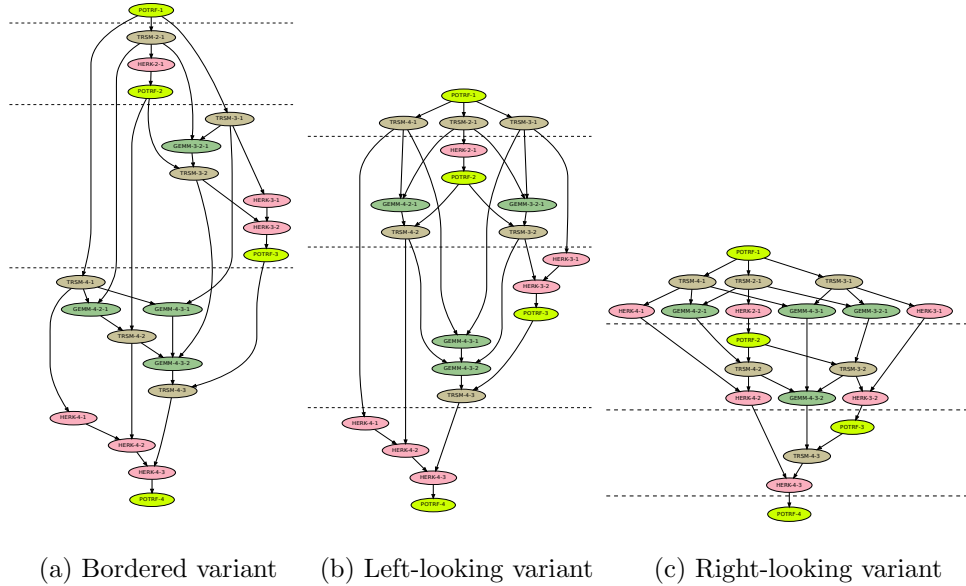


Figure 1.3: Three variants of the Cholesky decomposition applied to a tiled matrix of 4×4 tiles.

The first observation that one makes is that the height of each DAG varies according to which variant is chosen. The bordered variant is the tallest since the tasks

become almost sequential where the only portion that is not sequential is that of the row block update. The left-looking variant is almost of height t^2 where t is the number of tiles in a column of the tiled matrix. It gains parallelism from being able to update the column block of the final step within the loop in parallel. As before, the right-looking variant is the shortest and provides the most parallelism.

However, in the tiled versions, the synchronizations between each step of the LAPACK algorithms is superficial and can be removed. By doing so, these three variants reduce to only the right-looking variant.

The main difference between the tiled version and the blocked version is the amount of parallelism that is gained from updates of the trailing matrix. Instead of performing a large symmetric rank k update where k is the number of rows in a row block or the number of columns in a column block, the operation is decomposed into smaller symmetric rank n_b updates and associated matrix multiplications where n_b is the size of a tile such that $N = t \cdot n_b$. In the right-looking variant, for an $N \times N$ matrix the size of the first trailing matrix is $(N - k) \times (N - k)$ so that the update operation for this first matrix has a computational cost of $O(kN^2)$. The tiled update consists of both symmetric updates and matrix multiplications of tiles of size $n_b \times n_b$ so that the computational cost per task is $O(n_b^3)$.

The size of the tiles will determine the granularity of the tasks for a tiled algorithm. For a matrix of size $N \times N$, the tile size of the tiled $t \times t$ matrix can vary from the entire size of the matrix ($t = 1$) down to a scalar ($t = N$), but is held constant throughout the execution of the algorithm. However, a balance must be kept between the efficiency of the kernel and the amount of data movement.

Therefore, a tiled algorithm does overcome the restriction on the granularity imposed by the panels concept of LAPACK as well as alleviate some of the synchronizations and associated overhead. The memory bound is still present due to needing to move the updates of the trailing matrices.

The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [2] library provides the framework for the tiled algorithms. For the experimental portions, our main assumption will be a shared memory machine architecture wherein each processor has direct access to all portions of the memory in which the matrices are stored.

In Chapter 2 we describe more fully the implementation of the tiled Cholesky decomposition and further the tiled Cholesky Inversion algorithm. It shows that translating from LAPACK to PLASMA can be straight forward, but that there are caveats to be taken into account. In Chapter 3 we have implemented a tiled QR decomposition showing that the implementation is not straight-forward. Moreover, results from previous work do not translate directly to the tiled algorithm.

Unlike Chapters 2 and 3 where an unbounded number of processors is assumed, Chapters 4 and 5 restrict the number of processors and provide bounds on the performance of the algorithms. We observe the theoretical speed-up and efficiency and provide more realistic bounds on the performance.

Finally, Chapter 6 presents a study on a tiled implementation of the Strassen-Winograd algorithm for matrix-matrix multiplication. Unlike the other algorithms presented, it is a recursive algorithm which becomes interesting in the scope of tiled matrices.

2. Cholesky Inversion

In this chapter, we present joint work with Emmanuel Agullo, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg [4].

The appropriate direct method to compute the solution of a symmetric positive definite system of linear equations consists of computing the Cholesky factorization of that matrix and then solving the underlying triangular systems. It is not recommended to use the inverse of the matrix in this case. However some applications need to explicitly form the inverse of the matrix. A canonical example is the computation of the variance-covariance matrix in statistics. Higham [32, p.260,§3] lists more such applications.

With their advent, multicore architectures [50] induce the need for algorithms and libraries that fully exploit their capacities. A class of such algorithms – called tile algorithms [18, 19] – has been developed for one-sided dense factorizations (Cholesky, LU and QR) and made available as part of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) library [2]. In this chapter, we extend this class of algorithms to the case of the (symmetric positive definite) matrix inversion. Besides constituting an important functionality for a library such as PLASMA, the study of the matrix inversion on multicore architectures represents a challenging algorithmic problem. Indeed, first, contrary to standalone one-sided factorizations that have been studied so far, the matrix inversion exhibits many anti-dependences [6] (Write after Read). This is a false or artificial dependency which is reliant on the name of the data and not the actual data flow. For example, given two operations where the first only reads the data in the matrix A and the second only writes to the location of A , then in a parallel execution there may be a case where the data being read by the first operation is wrong since the second may have already written to the location. By copying the data from A beforehand, both operations can be executed in parallel. Those anti-dependences can be a bottleneck for parallel processing, which

is critical on multicore architectures. It is thus essential to investigate (and adapt) well known techniques used in compilation such as using temporary copies of data to remove anti-dependences to enhance the degree of parallelism of the matrix inversion. This technique is known as *array renaming* [6] (or *array privatization* [28]). Second, *loop reversal* [6] is to be investigated. Third, the matrix inversion consists of three successive steps (first of which is the Cholesky decomposition). In terms of scheduling, it thus represents an opportunity to study the effects of *pipelining* [6] those steps on performance.

The current version of PLASMA (version 2.1) is scheduled statically. Initially developed for the IBM Cell processor [34], this static scheduling relies on POSIX threads and simple synchronization mechanisms. It has been designed to maximize data reuse and load balancing between cores, allowing for very high performance [3] on today's multicore architectures. However, in the case of matrix inversion, the design of an ad-hoc static scheduling is a time consuming task and raises load balancing issues that are much more difficult to address than for a stand-alone Cholesky decomposition, in particular when dealing with the pipelining of multiple steps. Furthermore, the growth of the number of cores and the more complex memory hierarchies make executions less deterministic. In this chapter, we rely on an experimental in-house dynamic scheduler [33]. This scheduler is based on the idea of expressing an algorithm through its sequential representation and unfolding it at runtime using data hazards (Read after Write, Write after Read, Write after Write) as constraints for parallel scheduling. The concept is rather old and has been validated by a few successful projects. We could have as well used schedulers from the Jade project from Stanford University [42] or from the SMPs project from the Barcelona Supercomputer Center [40].

Our discussions are illustrated with experiments conducted on a dual-socket quad-core machine based on an Intel Xeon EMT64 processor operating at 2.26 GHz. The

theoretical peak is equal to 9.0 Gflop/s per core or 72.3 Gflop/s for the whole machine, composed of 8 cores. The machine is running Mac OS X 10.6.2 and is shipped with the Apple vecLib v126.0 multithreaded BLAS [15] and LAPACK vendor library, as well as LAPACK [7] v3.2.1 and ScaLAPACK [13] v1.8.0 references.

2.1 Tile in-place matrix inversion

Tile algorithms are a class of Linear Algebra algorithms that allow for fine granularity parallelism and asynchronous scheduling, enabling high performance on multicore architectures [3, 18, 19, 41]. The matrix of order n is split into $t \times t$ square submatrices of order b ($n = b \times t$). Such a submatrix is of small granularity (we fixed $b = 200$ in this chapter) and is called a *tile*. So far, tile algorithms have been essentially used to implement one-sided factorizations [3, 18, 19, 41].

Algorithm 2.1 extends this class of algorithms to the case of the matrix inversion. As in state-of-the-art libraries (LAPACK, ScaLAPACK), the matrix inversion is performed *in-place*, *i.e.*, the data structure initially containing matrix A is directly updated as the algorithm is progressing, without using any significant temporary extra-storage; eventually, A^{-1} replaces A . Algorithm 2.1 is composed of three steps. Step 1 is a Tile Cholesky Factorization computing the Cholesky factor L (lower triangular matrix satisfying $A = LL^T$). This step was studied in [19]. Step 2 computes L^{-1} by inverting L . Step 3 finally computes the inverse matrix $A^{-1} = L^{-1T}L^{-1}$. Each step is composed of multiple fine granularity tasks (since operating on tiles). These tasks are part of the BLAS (SYRK, GEMM, TRSM, TRMM) and LAPACK (POTRF, TRTRI, LAUUM) standards. A more detailed description is beyond the scope of this extended chapter and is not essential to the understanding of the rest of the chapter. Indeed, from a high level point of view, an operation based on tile algorithms can be represented as a Directed Acyclic Graphs (DAG) [22] where nodes represent the fine granularity tasks in which the operation can be decomposed and the edges represent the dependences among them. For instance, Figure 2.1a represents

Algorithm 2.1: Tile In-place Cholesky Inversion (lower format). Matrix A is the on-going updated matrix (in-place algorithm).

Input: A , Symmetric Positive Definite matrix in tile storage ($t \times t$ tiles).

Result: A^{-1} , stored in-place in A .

```

1  Step 1: Tile Cholesky Factorization (compute  $L$  such that  $A = LL^T$ );
2  for  $j = 0$  to  $t - 1$  do
3      for  $k = 0$  to  $j - 1$  do
4           $A_{j,j} \leftarrow A_{j,j} - A_{j,k} * A_{j,k}^T$  (SYRK(j,k)) ;
5       $A_{j,j} \leftarrow CHOL(A_{j,j})$  (POTRF(j)) ;
6      for  $i = j + 1$  to  $t - 1$  do
7          for  $k = 0$  to  $j - 1$  do
8               $A_{i,j} \leftarrow A_{i,j} - A_{i,k} * A_{j,k}^T$  (GEMM(i,j,k)) ;
9          for  $i = j + 1$  to  $t - 1$  do
10              $A_{i,j} \leftarrow A_{i,j} / A_{j,j}^T$  (TRSM(i,j)) ;
11 Step 2: Tile Triangular Inversion of  $L$  (compute  $L^{-1}$ );
12 for  $j = t - 1$  to  $0$  do
13      $A_{j,j} \leftarrow TRINV(A_{j,j})$  (TRTRI(j)) ;
14     for  $i = t - 1$  to  $j + 1$  do
15          $A_{i,j} \leftarrow A_{i,i} * A_{i,j}$  (TRMM(i,j)) ;
16         for  $k = j + 1$  to  $i - 1$  do
17              $A_{i,j} \leftarrow A_{i,j} + A_{i,k} * A_{k,j}$  (GEMM(i,j,k)) ;
18          $A_{i,j} \leftarrow -A_{i,j} * A_{i,i}$  (TRMM(i,j)) ;
19 Step 3: Tile Product of Lower Triangular Matrices (compute  $A^{-1} = L^{-1T} L^{-1}$ );
20 for  $i = 0$  to  $t - 1$  do
21     for  $j = 0$  to  $i - 1$  do
22          $A_{i,j} \leftarrow A_{i,i}^T * A_{i,j}$  (TRMM(i,j)) ;
23      $A_{i,i} \leftarrow A_{i,i}^T * A_{i,i}$  (LAUUM(i)) ;
24     for  $j = 0$  to  $i - 1$  do
25         for  $k = i + 1$  to  $t - 1$  do
26              $A_{i,j} \leftarrow A_{i,j} + A_{k,i}^T * A_{k,j}$  (GEMM(i,j,k)) ;
27     for  $k = i + 1$  to  $t - 1$  do
28          $A_{i,i} \leftarrow A_{i,i} + A_{k,i}^T * A_{k,i}$  (SYRK(i,k)) ;

```

(a) In-place (Algorithm 2.1)

(b) Out-of-place (variant introduced in § 2.2)

Algorithm 2.1 is based on the variants used in LAPACK 3.2.1. Bientinesi, Gunter and van de Geijn [10] discuss the merits of algorithmic variations in the case of the computation of the inverse of a symmetric positive definite matrix. Although of definite interest, this is not the focus of this extended chapter.

12

Table 2.1: Length of the critical path as a function of the number of tiles t .

	In-place case	Out-of-place case
Step 1	$3t - 2$	$3t - 2$
Step 2	$3t - 3$	$2t - 1$
Step 3	$3t - 2$	t

scalability than the blocked versions, similarly to what had been observed for the factorization step [3, 18, 19, 41].

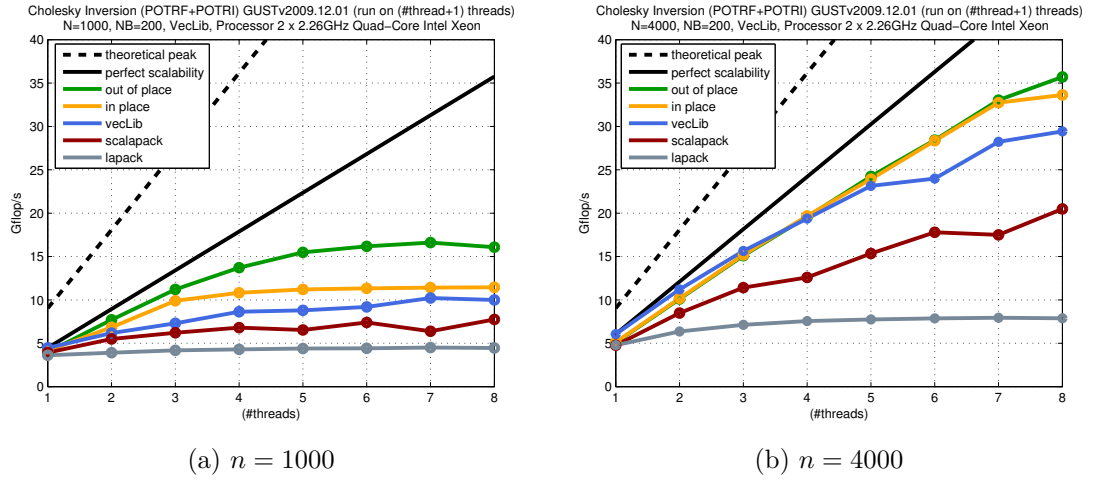


Figure 2.2: Scalability of Algorithm 2.1 (in place) and its out-of-place variant introduced in § 2.2, using our dynamic scheduler against vecLib, ScaLAPACK and LAPACK libraries.

2.2 Algorithmic study

In the § 2.1, we compared the performance of the tile Cholesky inversion against state-the-art libraries. In this section, we focus on tile Cholesky inversion and we discuss the impact of several variants of Algorithm 2.1 on performance.

Array renaming (removing anti-dependences). The dependence between SYRK(0,1) and TRMM(1,0) in the DAG of Step 3 of Algorithm 2.1 (Figure 2.1a) represents the constraint that the SYRK operation (l. 28 of Algorithm 2.1) needs to read $A_{k,i} = A_{1,0}$ before TRMM (l. 22) can overwrite $A_{i,j} = A_{1,0}$. This anti-dependence (Write after Read) can be removed thanks to a temporary copy of $A_{1,0}$.

Similarly, all the SYRK-TRMM anti-dependences, as well as TRMM-LAUMM and GEMM-TRMM anti-dependences can be removed. We have designed a variant of Algorithm 2.1 that removes all the anti-dependences thanks to the use of a large working array (this technique is called *array renaming* [6] in compilation [6]). The subsequent DAG (Figure 2.1b) is split in multiple pieces (Figure 2.1b), leading to a shorter critical path (Table 2.1). We implemented the out-of-place algorithm, based on our dynamic scheduler too. Figure 2.2a shows that our dynamic scheduler exploits its higher degree of parallelism to achieve a much higher strong scalability even on small matrices ($N = 1000$). For a larger matrix (Figure 2.2b), the in-place algorithm already achieved very good scalability. Therefore, using up to 7 cores, their performance are similar. However, there is not enough parallelism with a 4000×4000 matrix to use efficiently all 8 cores with the in-place algorithm; thus the higher performance of the out-of-place version in this case (leftmost part of Figure 2.2b).

Loop reversal (exploiting commutativity). The most internal loop of each step of Algorithm 2.1 (l. 8, l. 17 and l. 26) consists in successive commutative GEMM operations. Therefore they can be performed in any order, among which increasing order and decreasing order of the loop index. Their ordering impacts the length of the critical path. Algorithm 2.1 orders those three loops in increasing (U) and decreasing (D) order, respectively. We had manually chosen these respective orders (UDU) because they minimize the critical path of each step (values reported in Table 2.1). A naive approach would have, for example, been comprised of consistently ordering the loops in increasing order (UUU). In this case (UUU), the critical path of TRTRI would have been equal to $t^2 - 2t + 3$ (in-place) or $(\frac{1}{2}t^2 - \frac{1}{2}t + 2)$ (out-of-place) instead of $3t - 3$ (in-place) or $2t - 1$ (out-of-place) for (UDU). Figure 2.3 shows how loop reversal impacts performance.

Pipelining. Pipelining the multiple steps of the inversion reduces the length of its critical path. For the in-place case, the critical path is reduced from $9t - 7$ tasks

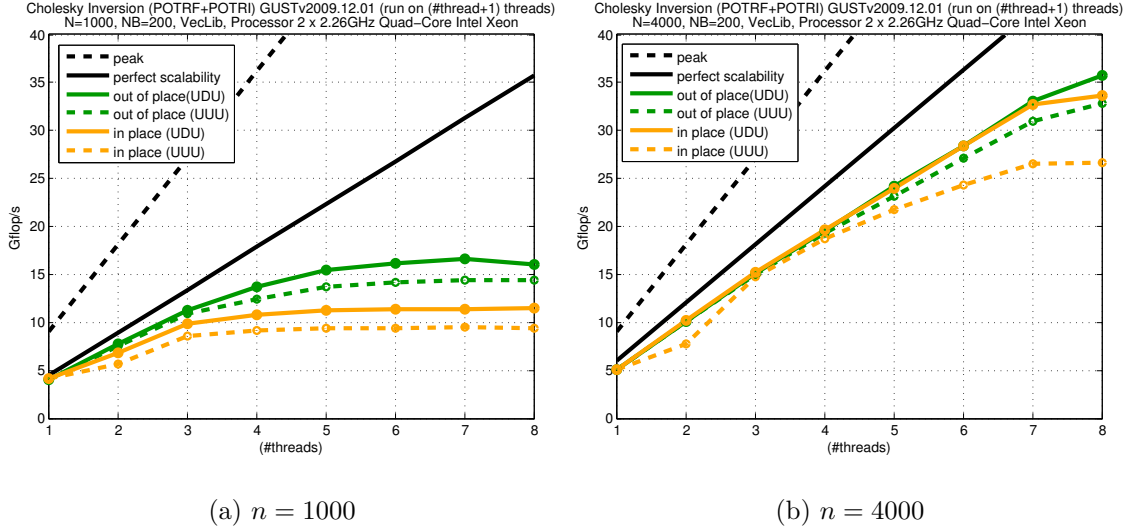


Figure 2.3: Impact of loop reversal on performance.

(t is the number of tiles) to $9t - 9$ tasks (negligible). For the out-of-place case, it is reduced from $6t - 3$ to $5t - 2$ tasks. We studied the effect of pipelining on the performance of the inversion on a 8000×8000 matrix with an artificially large tile size ($b = 2000$ and $t = 4$). As expected, we observed almost no effect on performance of the in-place case (about 36.4 seconds with or without pipelining). For the out-of-place case, the elapsed time grows from 25.1 to 29.2 seconds (16% overhead) when pipelining is prevented.

2.3 Conclusion and future work

We have proposed a new algorithm to compute the inverse of a symmetric positive definite matrix on multicore architectures. An experimental study has shown both an excellent scalability of our algorithm and a significant performance improvement compared to state-of-the-art libraries. Beyond extending the class of so-called tile algorithms, this study brought back to the fore well known issues in the domain of compilation. Indeed, we have shown the importance of loop reversal, array renaming and pipelining.

The use of a dynamic scheduler allowed an out-of-the-box pipeline of the different steps whereas loop reversal and array renaming required a manual change to the

algorithm. The future work directions consist in enabling the scheduler to perform itself loop reversal and array renaming. We exploited the commutativity of GEMM operations to perform array renaming. Their associativity would furthermore allow to process them in parallel (following a binary tree); the subsequent impact on performance is to be studied. Array renaming requires extra-memory. It will be interesting to address the problem of the maximization of performance under memory constraint. This work aims to be incorporated into PLASMA.

3. QR Factorization

In this chapter we present joint work with Mathias Jacquelin, Julien Langou, and Yves Robert [31].

Given an m -by- n matrix A with $n \leq m$, we consider the computation of its QR factorization, which is the factorization $A = QR$, where Q is an m -by- n unitary matrix ($Q^H Q = I_n$), and R is upper triangular.

The QR factorization of an m -by- n matrix with $n \leq m$ is the time consuming stage of some important numerical computations. It is needed for solving a linear least squares problem with m equations (observations) and n unknowns and is used to compute an orthonormal basis (the Q -factor) of the column span of the initial matrix A . For example, all block iterative methods (used to solve large sparse linear systems of equations or computing some relevant eigenvalues of such systems) require orthogonalizing a set of vectors at each step of the process. For these two usage examples, while $n \leq m$, n can range from $n \ll m$ to $n = m$. We note that the extreme case $n = m$ is also relevant: the QR factorization of a matrix can be used to solve (square) linear systems of equations. While this requires twice as many flops as an LU factorization, using a QR factorization (a) is unconditionally stable (Gaussian elimination with partial pivoting or pairwise pivoting is not) and (b) avoids pivoting so it may well be faster in some cases.

To obtain a QR factorization, we consider algorithms which apply a sequence of m -by- m unitary transformations, U_i , ($U_i^H U_i = I$), $i = 1, \dots, \ell$, on the left of the matrix A , such that after ℓ transformations the resulting matrix $R = U_\ell \dots U_1 A$ is upper triangular, in which case, R is indeed the R -factor of the QR factorization. The Q -factor (if needed) can then be obtained by computing $Q = U_1^H \dots U_\ell^H$. These types of algorithms are in regular use, e.g., in the LAPACK and ScaLAPACK libraries, and are favored over others algorithms (Cholesky QR or Gram-Schmidt) for their stability.

The unitary transformation U_i is chosen so as to introduce some zeros in the current update matrix $U_{i-1} \dots U_1 A$. The two basic transformations are Givens rotations and Householder reflections. One Givens rotation introduces one additional zero; the whole triangularization requires $mn - n(n+1)/2$ Givens rotations for $n < m$. One elementary Householder reflection simultaneously introduces $m - i$ zeros in position $i + 1$ to m in column i ; the whole triangularization requires n Householder reflections for $n < m$. (See LAPACK subroutine *GEQR2*.) The LAPACK *GEQRT* subroutine constructs a compact WY representation to apply a sequence of i_b Householder reflections, this enables one to introduce the appropriate zeros in i_b consecutive columns and thus leverage optimized Level 3 BLAS subroutines during the update. The blocking of Givens rotations is also possible but is more costly in terms of flops.

The main interest of Givens rotations over Householder transformations is that one can concurrently introduce zeros using disjoint pairs of rows, in other words, two transformations U_i and U_{i+1} may be applicable concurrently. This is not possible using the original Householder reflection algorithm since the transformations work on whole columns and thus do not exhibit this type of intrinsic parallelism, forcing this kind of Householder reflections to be applied sequentially. The advantages of Householder reflections over Givens rotations are that, first, Householder reflections perform fewer flops, and second, the compact WY transformation enables high sequential performance of the algorithm. In a multicore setting, where data locality and parallelism are crucial algorithmic characteristics for enabling performance, the tiled QR factorization algorithm combines both ideas: use of Householder reflections for high sequential performance and use of a scheme such as Givens rotations to enable parallelism within cores. In essence, one can think either (i) of the tiled QR factorization as a Givens rotation scheme but on tiles (m_b -by- n_b submatrices) instead of on scalars (1-by-1 submatrices) as in the original scheme, or (ii) of it as a blocked Householder reflection scheme where each reflection is confined to an extent much

less than the full column span, which enables concurrency with other reflections.

Tiled QR factorization in the context of multicore architectures has been introduced in [18, 19, 41]. Initially the focus was on square matrices and the sequence of unitary transformations presented was analogous to SAMEH-KUCK [45], which corresponds to reducing the panels with flat trees. The possibility of using any tree in order to either maximize parallelism or minimize communication is explained in [26]. The focus of this chapter is on maximizing parallelism. We reduce the communication (data movement between memory hierarchy) within the algorithm to acceptable levels by tiling the operations. Stemming from the observation that a binary tree is best for tall and skinny matrices and a flat tree is best for square matrices, Hadri et al. [30], propose to use trees which combine flat trees at the bottom level with a binary tree at the top level in order to exhibit more parallelism. Our theoretical and experimental work explains that we can adapt FIBONACCI [36] and GREEDY [24, 25] to tiles, resulting in yet better algorithms in terms of parallelism. Moreover our new algorithms do not have any tuning parameter such as the domain size in the case of [30].

The sequential kernels of the Tiled QR factorization (executed on a core) are made of standard blocked algorithms such as LAPACK encoded in kernels; the development of these kernels is well understood. The focus of this chapter is on improving the overall degree of parallelism of the algorithm. Given a p -by- q tiled matrix, we seek to find an appropriate sequence of unitary transformations on the tiled matrix so as to maximize parallelism (minimize critical path length). We will get our inspiration from previous work from the 1970s/80s on Givens rotations where the question was somewhat related: given an m -by- n matrix, find an appropriate sequence of Givens rotations as to maximize parallelism. This question is essentially answered in [24, 25, 36, 45]; we call this class of algorithms “*coarse-grain algorithms*.”

Working with tiles instead of scalars, we introduce four essential differences be-

tween the analysis and the reality of the tiled algorithms versus the coarse-grain algorithms. First, while there are only two states for a scalar (nonzero or zero), a tile can be in three states (zero, triangle or full). Second, there are more operations available on tiles to introduce zeros; we have a total of three different tasks which can introduce zeros in a matrix. Third, in the tiled algorithms, the factorization and the update are dissociated to enable factorization stages to overlap with update stages; whereas, in the coarse-grain algorithm, the factorization and the associated update are considered as a single stage. Lastly, while coarse-grain algorithms have only one task, we end up with six different tasks: three from the factorizations (zeroing of tiles) and three for each of the associated updates (since these have been unlinked from the factorization). Each of these six tasks have different computational weights; this dramatically complicates the critical path analysis of the tiled algorithms.

While the GREEDY algorithm is optimal for coarse-grain algorithms, we show that it is not in the case of tiled algorithms. However, we have devised and proved that there does exist an optimal tiled algorithm.

3.1 The QR factorization algorithm

Tiled algorithms are expressed in terms of tile operations rather than elementary operations. Each tile is of size $n_b \times n_b$, where n_b is a parameter tuned to squeeze the most out of arithmetic units and memory hierarchy. Typically, n_b ranges from 80 to 200 on state-of-the-art machines [5]. Algorithm 3.1 outlines a naive tiled QR algorithm, where loop indices represent tiles:

Algorithm 3.1: Generic QR algorithm for a tiled $p \times q$ matrix.

```

1 for  $k = 1$  to  $\min(p, q)$  do
2   forall the  $i \in \{k + 1, \dots, p\}$  using any ordering, do
3      $\text{elim}(i, \text{piv}(i, k), k)$ 

```

In Algorithm 3.1, k is the panel index, and $\text{elim}(i, \text{piv}(i, k), k)$ is an orthogonal transformation that combines rows i and $\text{piv}(i, k)$ to zero out the tile in position (i, k) .

However, this formulation is somewhat misleading, as there is much more freedom for QR factorization algorithms than, say, for Cholesky algorithms (and contrarily to LU elimination algorithms, there are no numerical stability issues). For instance in column 1, the algorithm must eliminate all tiles $(i, 1)$ where $i > 1$, but it can do so in several ways. Take $p = 6$. Algorithm 3.1 uses the transformations

$$\text{elim}(2, 1, 1), \text{elim}(3, 1, 1), \text{elim}(4, 1, 1), \text{elim}(5, 1, 1), \text{elim}(6, 1, 1)$$

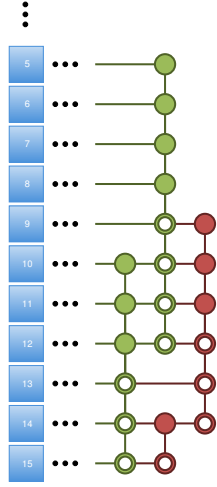
But the following scheme is also valid:

$$\text{elim}(3, 1, 1), \text{elim}(6, 4, 1), \text{elim}(2, 1, 1), \text{elim}(5, 4, 1), \text{elim}(4, 1, 1)$$

In this latter scheme, the first two transformations $\text{elim}(3, 1, 1)$ and $\text{elim}(6, 4, 1)$ use distinct pairs of rows, and they can execute in parallel. On the contrary, $\text{elim}(3, 1, 1)$ and $\text{elim}(2, 1, 1)$ use the same pivot row and must be sequentialized. To complicate matters, it is possible to have two orthogonal transformations that execute in parallel but involve zeroing a tile in two different columns. For instance we can add $\text{elim}(6, 5, 2)$ to the previous transformations and run it concurrently with, say, $\text{elim}(2, 1, 1)$. Any tiled QR algorithm will be characterized by an *elimination list*, which provides the ordered list of the transformations used to zero out all the tiles below the diagonal. This elimination list must obey certain conditions so that the factorization is valid. For instance, $\text{elim}(6, 5, 2)$ must follow $\text{elim}(6, 4, 1)$ and $\text{elim}(5, 4, 1)$ in the previous list, because there is a flow dependence between these transformations. Note that, although the elimination list is given as a totally ordered sequence, some transformations can execute in parallel, provided that they are not linked by a dependence: in the example, $\text{elim}(6, 4, 1)$ and $\text{elim}(2, 1, 1)$ could have been swapped, and the elimination list would still be valid.

In order to describe more fully the dependencies inherent in the eliminations we shall observe a snippet of an example. In Figure 3.1a, to the left we have the row identifications, the empty circles represent zeroed elements, and the filled circles

represent the pivots used to zero out the elements. The first column's eliminations are shown in green and the second in red. From the elimination list, we define $I_{s,k}$ as the set of rows in column k that are zeroed out at time step s .



(a) Diagram of elimination list

$$\begin{array}{l}
 \text{elim}(13, 10, 1) \\
 \text{elim}(14, 11, 1) \\
 \text{elim}(15, 12, 1) \\
 \text{elim}(9, 5, 1) \\
 \text{elim}(10, 6, 1) \\
 \text{elim}(11, 7, 1) \\
 \text{elim}(12, 8, 1) \\
 \text{elim}(15, 14, 2) \\
 \text{elim}(12, 9, 2) \\
 \text{elim}(13, 10, 2) \\
 \text{elim}(14, 11, 2)
 \end{array}
 \left\{
 \begin{array}{l}
 I_{s_1,1} \Rightarrow \text{elim}(I_{s_1,1}, 1) \\
 I_{s_2,1} \Rightarrow \text{elim}(I_{s_2,1}, 1) \\
 I_{s_1,2} \Rightarrow \text{elim}(I_{s_1,2}, 2) \\
 I_{s_2,2} \Rightarrow \text{elim}(I_{s_2,2}, 2)
 \end{array}
 \right.$$

(b) Elimination list

What may not be so evident from the elimination list but is more apparent in the diagram of the elimination list are the following dependency relationships (note that \prec indicates that the operation on the left must finish prior to the operation on the right starting):

$$\text{elim}(\text{piv}(I_{s,k}, k), k-1) \prec \text{elim}(I_{s,k}, k) \quad (3.1a)$$

$$\text{elim}(I_{s,k}, k-1) \prec \text{elim}(I_{s,k}, k) \quad (3.1b)$$

$$\text{elim}(I_{s-1,k}, k) \prec \text{elim}(I_{s,k}, k) \quad (3.1c)$$

However, not all of these dependencies may cause an elimination to be locked to a particular time step. In fact, some dependencies may not be needed for a particular instance, but the addition of these will not create an artificial lock. For example, $\text{elim}(I_{s_2,2}, 2)$ is dependent upon $\text{elim}(\text{piv}(I_{s_2,2}), 1)$, $\text{elim}(I_{s_2,2}, 1)$, and $\text{elim}(I_{s_1,2}, 2)$ but $\text{elim}(I_{s_1,2}, 2)$ only depends upon $\text{elim}(\text{piv}(I_{s_1,2}), 1)$ and $\text{elim}(I_{s_1,2}, 1)$.

Table 3.1: Kernels for tiled QR. The unit of time is $\frac{n_b^3}{3}$, where n_b is the blocksize.

Operation	Panel		Update	
	Name	Cost	Name	Cost
Factor square into triangle	<i>GEQRT</i>	4	<i>UNMQR</i>	6
Zero square with triangle on top	<i>TSQRT</i>	6	<i>TSMQR</i>	12
Zero triangle with triangle on top	<i>TTQRT</i>	2	<i>TTMQR</i>	6

Before formally stating the conditions that guarantee the validity of (the elimination list of) an algorithm, we explain how orthogonal transformations can be implemented.

3.1.1 Kernels

To implement a given orthogonal transformation $\text{elim}(i, \text{piv}(i, k), k)$, one can use six different kernels, whose costs are given in Table 3.1. In this table, the unit of time is the time to perform $\frac{n_b^3}{3}$ floating-point operations.

There are two main possibilities to implement an orthogonal transformation $\text{elim}(i, \text{piv}(i, k), k)$: The first version eliminates tile (i, k) with the *TS* (*Triangle on top of square*) kernels, as shown in Algorithm 3.2:

Algorithm 3.2: Elimination $\text{elim}(i, \text{piv}(i, k), k)$ via <i>TS</i> (<i>Triangle on top of square</i>) kernels.	
<hr/>	
1	<i>GEQRT</i> ($\text{piv}(i, k), k$)
2	<i>TSQRT</i> ($i, \text{piv}(i, k), k$)
3	for $j = k + 1$ to q do
4	<i>UNMQR</i> ($\text{piv}(i, k), k, j$)
5	<i>TSMQR</i> ($i, \text{piv}(i, k), k, j$)

Here the tile panel $(\text{piv}(i, k), k)$ is factored into a triangle (with *GEQRT*). The transformation is applied to subsequent tiles $(\text{piv}(i, k), j)$, $j > k$, in row $\text{piv}(i, k)$ (with *UNMQR*). Tile (i, k) is zeroed out (with *TSQRT*), and subsequent tiles (i, j) , $j > k$, in row i are updated (with *TSMQR*). The flop count is $4 + 6 + (6 + 12)(q - k) =$

$10 + 18(q - k)$ (expressed in same time unit as in Table 3.1). Dependencies are the following:

$$\begin{aligned}
GEQRT(piv(i, k), k) &\prec TSQRT(i, piv(i, k), k) \\
GEQRT(piv(i, k), k) &\prec UNMQR(piv(i, k), k, j) \quad \text{for } j > k \\
UNMQR(piv(i, k), k, j) &\prec TSMQR(i, piv(i, k), k, j) \quad \text{for } j > k \\
TSQRT(i, piv(i, k), k) &\prec TSMQR(i, piv(i, k), k, j) \quad \text{for } j > k
\end{aligned}$$

$TSQRT(i, piv(i, k), k)$ and $UNMQR(piv(i, k), k, j)$ can be executed in parallel, as well as $UNMQR$ operations on different columns $j, j' > k$. With an unbounded number of processors, the parallel time is thus $4 + 6 + 12 = 22$ time-units.

Algorithm 3.3: Elimination $elim(i, piv(i, k), k)$ via TT (*Triangle on top of triangle*) kernels.

```

1 if  $k > 0$  then
2    $\lfloor TTQRT(i, piv(i, k), k)$ 
3 if  $k < q$  then
4   for  $j = k + 1$  to  $q$  do
5     if  $k > 0$  then
6        $\lfloor TTMQR(i, piv(i, k), k, j)$ 
7    $GEQRT(i, k + 1)$ 
8   for  $j = k + 2$  to  $q$  do
9      $\lfloor UNMQR(i, k, j)$ 

```

The second approach to implement the orthogonal transformation $elim(i, piv(i, k), k)$ is with the TT (*Triangle on top of triangle*) kernels, as shown in Algorithm 3.3. Here tile (i, k) is zeroed out (with $TTQRT$) and subsequent tiles (i, j) and $(piv(i, k), j)$, $j > k$, in rows i and $piv(i, k)$ are updated (with $TTMQR$). Immediately following, tile $(i, k + 1)$ is factored into a triangle and the corresponding transformations are applied to the remaining columns in row i . Necessarily, $TTQRT$ must have the triangularization of tile (i, k) and $(piv(i, k), k)$ completed

in order to proceed. Hence for the first column there are no updates to be applied from previous columns such that the triangularization of these tiles (with *GEQRT*) is completed and can be considered a preprocessing step. The flop count is $2(4 + 6(q - k)) + 2 + 6(q - k) = 10 + 18(q - k)$, just as before. Dependencies are the following:

$$GEQRT(piv(i, k), k) \prec UNMQR(piv(i, k), k, j) \quad \text{for } j > k \quad (3.2a)$$

$$GEQRT(i, k) \prec UNMQR(i, k, j) \quad \text{for } j > k \quad (3.2b)$$

$$GEQRT(piv(i, k), k) \prec TTQRT(i, piv(i, k), k) \quad (3.2c)$$

$$GEQRT(i, k) \prec TTQRT(i, piv(i, k), k) \quad (3.2d)$$

$$TTQRT(i, piv(i, k), k) \prec TTMQR(i, piv(i, k), k, j) \quad \text{for } j > k \quad (3.2e)$$

$$UNMQR(piv(i, k), k, j) \prec TTMQR(i, piv(i, k), k, j) \quad \text{for } j > k \quad (3.2f)$$

$$UNMQR(i, k, j) \prec TTMQR(i, piv(i, k), k, j) \quad \text{for } j > k \quad (3.2g)$$

Now the factor operations in row $piv(i, k)$ and i can be executed in parallel. Moreover, the *UNMQR* updates can be run in parallel with the *TTQRT* factorization. Thus, with an unbounded number of processors, the parallel time is $4+6+6 = 16$ time-units.

Recall our definition of the set $I_{s,k}$ to be the set of rows in column k that will be zeroed out at time step s in the coarse-grain algorithm. Thus the following dependencies are a direct consequence of 3.1c as applied to the zeroing of a tile and the corresponding updates.

$$TTQRT(I_{s-1}, piv(I_{s-1}, k), k) \prec TTQRT(I_s, piv(I_s, k), k) \quad (3.3a)$$

$$TTQRT(I_{s-1}, piv(I_{s-1}, k), k, j) \prec TTMQR(I_s, piv(I_s, k), k, j) \quad \text{for } j > k \quad (3.3b)$$

In Algorithm 3.2 and 3.3, it is understood that if a tile is already in triangular form, then the associated *GEQRT* and update kernels do not need to be applied.

All the new algorithms introduced in this chapter are based on *TT* kernels. From an algorithmic perspective, *TT* kernels are more appealing than *TS* kernels, as they

offer more parallelism. More precisely, we can always break a TS kernel into two TT kernels: We can replace a $TSQRT(i, piv(i, k), k)$ (following a $GEQRT(piv(i, k), k)$) by a $GEQRT(i, k)$ and a $TTQRT(i, piv(i, k), k)$. A similar transformation can be made for the updates. Hence a TS -based tiled algorithm can always be executed with TT kernels, while the converse is not true. However, the TS kernels provide more data locality, they benefit from a very efficient implementation (see §3.3), and several existing algorithms use these kernels. For all these reasons, and for comprehensiveness, our experiments will compare approaches based on both kernel types.

At this point, the PLASMA library only contains TS kernels. We have mapped the PLASMA algorithm to TT kernel algorithm using this conversion. Going from a TS kernel algorithm to a TT kernel algorithm is implicitly done by Hadri et al. [11] when going from their “Semi-Parallel” to their “Fully-Parallel” algorithms.

3.1.2 Elimination lists

As stated above, any algorithm factorizing a tiled matrix of size $p \times q$ is characterized by its elimination list. Obviously, the algorithm must zero out all tiles below the diagonal: for each tile (i, k) , $i > k$, $1 \leq k \leq \min(p, q)$, the list must contain exactly one entry $elim(i, \star, k)$, where \star denotes some row index $piv(i, k)$. There are two conditions for a transformation $elim(i, piv(i, k), k)$ to be valid:

- both rows i and $piv(i, k)$ must be ready, meaning that all their tiles left of the panel (of indices (i, k') and $(piv(i, k), k')$ for $1 \leq k' < k$) must have already been zeroed out: all transformations $elim(i, piv(i, k'), k')$ and $elim(piv(i, k), piv(piv(i, k), k'), k')$ must precede $elim(i, piv(i, k), k)$ in the elimination list
- row $piv(i, k)$ must be a potential annihilator, meaning that tile $(piv(i, k), k)$ has not been zeroed out yet: the transformation $elim(piv(i, k), piv(piv(i, k), k), k)$ must follow $elim(i, piv(i, k), k)$ in the elimination list

Any algorithm that factorizes the tiled matrix obeying these conditions is called a *generic tiled algorithm* in the following.

Theorem 3.1 *No matter what elimination list (any combination of TT , TS) is used the total weight of the tasks for performing a tiled QR factorization algorithm is constant and equal to $6pq^2 - 2q^3$.*

Proof: We have that the quantity of each kernel is given by the following

$$L_1 :: GEQRT = TTQRT + q$$

$$L_2 :: UNMQR = TTMQR + (1/2)q(q - 1)$$

$$L_3 :: TTQRT + TSQRT = pq - (1/2)q(q + 1)$$

$$L_4 :: TTMQR + TSMQR = (1/2)pq(q - 1) - (1/6)q(q - 1)(q + 1)$$

The quantity of $TTQRT$ provides the number of tiles zeroed out via a triangle on top of a triangle kernel. Thus equation L_1 is composed of two parts: necessarily, the diagonal tiles must be triangularized and each $TTQRT$ must admit one more triangularization in order to provide a pairing. The number of updates of these triangularizations, given by L_2 , is simply the sum of the updates from the triangularization of the tiles on the diagonal and the updates from the zeroed tiles via $TTQRT$. The combination of $TTQRT$ and $TSQRT$, equation L_3 , is exactly the total number of tiles that are zeroed, namely every tile below the diagonal. Hence, the total number of updates, provided by L_4 , is the number of tiles below the diagonal beyond the first column minus the sum of the progression through the columns. Now we define

$$L_5 = 4L_1 + 6L_2 + 6L_3 + 12L_4$$

then

$$L_5 = 4GEQRT + 6TSQRT + 2TTQRT + 6UNMQR + 6TTMQR + 12TSMQR.$$

As can be noted in L_5 , the coefficients of each term correspond precisely to the weight of the kernels as derived from the number of flops each kernel incurs. Simplifying L_5 ,

we have our result

$$L_5 = 6pq^2 - 2q^3.$$

■

A critical result of Theorem 3.1 is that no matter what elimination list is used, the total weight of the tasks for performing a tiled QR factorization algorithm is constant and by using our unit task weight of $n_b^3/3$, with $m = pn_b$, and $n = qn_b$, we obtain $2mn^2 - 2/3n^3$ flops which is the exact same number as for a standard Householder reflection algorithm as found in LAPACK (e.g., [14]).

3.1.3 Execution schemes

In essence, the execution of a generic tiled algorithm is fully determined by its elimination list. This list is statically given as input to the scheduler, and the execution progresses dynamically, with the scheduler executing all required transformations as soon as possible. More precisely, each transformation involves several kernels, whose execution starts as soon as they are ready, i.e., as soon as all dependencies have been enforced. Recall that a tile (i, k) can be zeroed out only after all tiles (i, k') , with $k' < k$, have been zeroed out. Execution progresses as follows:

- Before being ready for elimination, tile (i, k) , $i > k$, must be updated $k-1$ times, in order to zero out the $k-1$ tiles to its left (of index (i, k') , $k' < k$). The last update is a transformation $TTMQR(i, piv(i, k-1), k-1, k)$ for some row index $piv(i, k-1)$ such that $elim(i, piv(i, k-1), k-1)$ belongs to the elimination list. When completed, this transformation triggers the transformation $GEQRT(i, k)$, which can be executed immediately after the completion of the $TTMQR$. In turn, $GEQRT(i, k)$ triggers all updates $UNMQR(i, k, j)$ for all $j > k$. These updates are executed as soon as they are ready for execution.
- The elimination $elim(i, piv(i, k), k)$ is performed as soon as possible when both rows i and $piv(i, k)$ are ready. Just after the completion of $GEQRT(i, k)$ and

$GEQRT(piv(i, k), k)$, kernel $TTQRT(i, piv(i, k), k)$ is launched. When finished, it triggers the updates $TTMQR(i, piv(i, k), k, j)$ for all $j > k$.

Obviously, the degree of parallelism that can be achieved depends upon the eliminations that are chosen. For instance, if all eliminations in a given column use the same factor tile, they will be sequentialized. This corresponds to the flat tree elimination scheme described below: in each column k , it uses $elim(i, k, k)$ for all $i > k$. On the contrary, two eliminations $elim(i, piv(i, k), k)$ and $elim(i', piv(i', k), k)$ in the same column can be fully parallelized provided that they involve four different rows. Finally, note that several eliminations can be initiated in different columns simultaneously, provided that they involve different pairs of rows, and that all these rows are ready (i.e., they have the desired number of leftmost zeros).

The following lemma will prove very useful; it states that we can assume w.l.o.g. that each tile is zeroed out by a tile above it, closer to the diagonal.

Lemma 3.2 *Any generic tiled algorithm can be modified, without changing its execution time, so that all eliminations $elim(i, piv(i, k), k)$ satisfy $i > piv(i, k)$.*

Proof: Define a *reverse* elimination as an elimination $elim(i, piv(i, k), k)$ where $i < piv(i, k)$. Consider a generic tiled algorithm whose elimination list contains some reverse eliminations. Let k_0 be the first column to contain one of them. Let i_0 be the largest row index involved in a reverse elimination in column k_0 . The elimination list in column k_0 may contain several reverse eliminations $elim(i_1, i_0, k_0)$, $elim(i_2, i_0, k_0)$, \dots , $elim(i_r, i_0, k_0)$, in that order, before row i_0 is eventually zeroed out by the transformation $elim(i_0, piv(i_0, k_0), k_0)$. Note that $piv(i_0, k_0) < i_0$ by definition of i_0 . We modify the algorithm by exchanging the roles of rows i_0 and i_1 in column k_0 : the elimination list now includes $elim(i_0, i_1, k_0)$, $elim(i_2, i_1, k_0)$, \dots , $elim(i_r, i_1, k_0)$, and $elim(i_1, piv(i_0, k_0), k_0)$. All dependencies are preserved, and the execution time is unchanged. Now the largest row index involved in a reverse elimination in column k_0

is strictly smaller than i_0 , and we repeat the procedure until there does not remain any reverse elimination in column k_0 . We proceed inductively to the following columns, until all reverse eliminations have been suppressed. ■

3.2 Critical paths

In this section we describe several generic tiled algorithms, and we provide their critical paths, as well as optimality results. These algorithms are inspired by algorithms that have been introduced twenty to thirty years ago [45, 36, 25, 24], albeit for a much simpler, *coarse-grain* model. In this “old” model, the time-unit is the time needed to execute an orthogonal transformation across two matrix rows, regardless of the position of the zero to be created, hence regardless of the length of these rows. Although the granularity is much coarser in this model, any existing algorithm for the old model can be transformed into a generic tiled algorithm, just by enforcing the very same elimination list provided by the algorithm. Critical paths are obtained using a discrete event based simulator specially developed to this end, based on the Simgrid framework [47]. It carefully handles dependencies across tiles, and allows for the analysis of both static and dynamic algorithms.¹

3.2.1 Coarse-grain algorithms

We start with a short description of three algorithms for the coarse-grain model. These algorithms are illustrated in Table 3.2 for a 15×6 matrix.

3.2.1.1 SAMEH-KUCK algorithm

The SAMEH-KUCK algorithm [45] uses the panel row for all eliminations in each column, starting from below the diagonal and proceeding downwards. Time-steps indicate the time-unit at which the elimination can be done, assuming unbounded resources. Formally, the elimination list is

$$\{(elim(i, k, k), i = k + 1, k + 2, \dots, p), k = 1, 2, \dots, \min(p, q)\}$$

¹The discrete event based simulator, together with the code for all tiled algorithms, is publicly available at <http://graal.ens-lyon.fr/~mjacquel/tiledQR.html>

This algorithm is also referred as FLATTREE.

3.2.1.2 FIBONACCI algorithm

The FIBONACCI algorithm is the Fibonacci scheme of order 1 in [36]. Let $coarse(i, k)$ be the time-step at which tile (i, k) , $i > k$, is zeroed out. These values are computed as follows. In the first column, there are one 5, two 4's, three 3's, four 2's and four 1's (we would have had five 1's with $p = 16$). Given x as the least integer such that $x(x + 1)/2 \geq p - 1$, we have $coarse(i, 1) = x - y + 1$ where y is the least integer such that $i \leq y(y + 1)/2 + 1$. Let the row indices of the z tiles that are zeroed out at step s , $1 \leq s \leq x$, range from i to $i + z - 1$. The elimination list for these tiles is $elim(i + j, piv(i + j, 1), 1)$, with $piv(i + j) = i + j - z$ for $0 \leq j \leq z - 1$. In other words, to eliminate a bunch of z consecutive tiles at the same time-step, the algorithm uses the z rows above them, pairing them in the natural order. Now the elimination scheme of the next column is the same as that of the previous column, shifted down by one row, and adding two time-units: $coarse(i, k) = coarse(i - 1, k - 1) + 2$, while the pairing obeys the same rule.

3.2.1.3 GREEDY algorithm

At each step, the GREEDY algorithm [24, 25] eliminates as many tiles as possible in each column, starting with bottom rows. The pairing for the eliminations is done exactly as for FIBONACCI. There is no closed-form formula to compute $coarse(i, k)$, the time-step at which tile (i, k) is eliminated, but it is possible to provide recursive expressions (see [24, 25]).

Consider a rectangular $p \times q$ matrix, with $p > q$. With the coarse-grain model, the critical path of SAMEH-KUCK is $p + q - 2$, and that of FIBONACCI is $x + 2q - 2$, where x is the least integer such that $x(x + 1)/2 \geq p - 1$. The critical path of GREEDY is unknown, but the critical path of GREEDY is optimal. For square $q \times q$ matrices, critical paths are slightly different ($2q - 3$ for SAMEH-KUCK, $x + 2q - 4$ for FIBONACCI).

Table 3.2: Time-steps for coarse-grain algorithms.

(a) SAMEH-KUCK							(b) FIBONACCI							(c) GREEDY						
★							★							★						
1	★						5	★						4	★					
2	3	★					4	7	★					3	6	★				
3	4	5	★				4	6	9	★				3	5	8	★			
4	5	6	7	★			3	6	8	11	★			2	5	7	10	★		
5	6	7	8	9	★		3	5	8	10	13	★		2	4	7	9	12	★	
6	7	8	9	10	11		3	5	7	10	12	15		2	4	6	9	11	14	
7	8	9	10	11	12		2	5	7	9	12	14		2	4	6	8	10	13	
8	9	10	11	12	13		2	4	7	9	11	14		1	3	5	8	10	12	
9	10	11	12	13	14		2	4	6	9	11	13		1	3	5	7	9	11	
10	11	12	13	14	15		2	4	6	8	11	13		1	3	5	7	9	11	
11	12	13	14	15	16		1	4	6	8	10	13		1	3	4	6	8	10	
12	13	14	15	16	17		1	3	6	8	10	12		1	2	4	6	8	10	
13	14	15	16	17	18		1	3	5	8	10	12		1	2	4	5	7	9	
14	15	16	17	18	19		1	3	5	7	10	12		1	2	3	5	6	8	

3.2.2 Tiled algorithms

As stated above, each coarse-grain algorithm can be transformed into a tiled algorithm, simply by keeping the same elimination list, and triggering the execution of each kernel as soon as possible. However, because the weights of the factor and update kernels are not the same, it is much more difficult to compute the critical paths of the transformed (tiled) algorithms. Table 3.3 is the counterpart of Table 3.2, and depicts the time-steps at which tiles are actually zeroed out. Note that the tiled version of SAMEH-KUCK is indeed the FLAT TREE algorithm in PLASMA [18, 19], and we have renamed it accordingly. As an example, Algorithm 3.4 shows the GREEDY algorithm for the tiled model.

A first (and quite unexpected) result is that GREEDY is no longer optimal, as shown in the first two columns of Table 3.3a for a 15×2 matrix. In each column and at each step, “*the ASAP algorithm*” starts the elimination of a tile as soon as there are at least two rows ready for the transformation. When $s \geq 2$ eliminations can start simultaneously, ASAP pairs the $2s$ rows just as FIBONACCI and GREEDY, the first row (closest to the diagonal) with row $s+1$, the second row with row $s+2$, and so on. As a

matter of a fact, when processing the second column, both ASAP and GREEDY begin with the elimination of lines 10 to 15 (at time step 20). However, once tiles $(13, 2)$, $(14, 2)$ and $(15, 2)$ are zeroed out (i.e. at time step 22), ASAP eliminates 4 zeros, in rows 9 through 12. On the contrary, GREEDY waits until time step 26 to eliminate 6 zeros in rows 6 through 12. In a sense, ASAP is the counterpart of GREEDY at the tile level. However, ASAP is not optimal either, as shown in Table 3.3a for a 15×3 matrix. On larger examples, the critical path of GREEDY is better than that of ASAP, as shown in Table 3.3b.

We can however use the optimality of the coarse-grain GREEDY to devise an optimal tiled algorithm. Let us define the following algorithm:

Definition 3.3 *Given a matrix of $p \times q$ tiles, with $p > q$, the GRASAP (i) algorithm*

1. *uses Algorithm 3.3 to execute GREEDY on the first $q - i$ columns and propagate the updates through column q .*
2. *and for column(s) $q - i + 1$ through q , apply the ASAP algorithm.*

Clearly, if we let $i = q$ we obtain the ASAP algorithm. We define GRASAP to be GRASAP (1), i.e., only the elimination of the last column will differ from GREEDY, and we will show that GRASAP is an optimal tiled algorithm.

Although we cannot provide an elimination list for the entire tiled matrix of size $p \times q$, we do provide an elimination list for the first $q - 1$ columns. This tiled elimination list describes the time-steps at which Algorithm 3.3 is complete, i.e., all of the factorization kernels are complete for $k \leq q - 1$ and corresponding update kernels are complete for all columns $k < j \leq q$.

We must make note of one consequence from the coarse-grain elimination list before proceeding. We will use this repeatedly within the proof of translating a coarse-grain elimination list to a tiled elimination list.

Lemma 3.4 *Given an elimination list from any coarse-grain algorithm, let $s = \text{coarse}(i, k)$ be the time step at which element (i, k) is eliminated and let*

$$I_{s,k} = \{i \mid s = \text{coarse}(I_{s,k}, k)\}.$$

Then for any s we have

$$s - 1 = \text{coarse}(I_{s,k}, k) - 1 \geq \max \left(\begin{array}{c} \text{coarse}(I_{s,k}, k - 1) \\ \text{coarse}(\text{piv}(I_{s,k}, k), k - 1) \end{array} \right)$$

and in particular

$$s_1 - 1 = \max \left(\begin{array}{c} \text{coarse}(I_{s_1,k}, k - 1) \\ \text{coarse}(\text{piv}(I_{s_1,k}, k), k - 1) \end{array} \right)$$

where $s_1 = \min_{k+1 \leq i \leq p}(\text{coarse}(i, k))$.

Proof: This follows directly from the dependencies given in (3.1a)-(3.1c). ■

Theorem 3.5 *Given the elimination list of a coarse-grain algorithm for a matrix of size $p \times q$, using Algorithm 3.3, the tiled elimination list for all but the last column is given by*

$$\text{tiled}(i, k) = 10k + 6 \cdot \text{coarse}(i, k), \quad 1 \leq i \leq p, 1 \leq k < q - 1$$

where $\text{coarse}(i, k)$ is the elimination list of the coarse-grain algorithm.

Proof: In this analysis, when k is clear, we will use I_s instead of $I_{s,k}$. By abuse of notation, we will write $\text{GEQRT}(i, k)$ to denote the time at which the task $\text{GEQRT}(i, k)$ is complete and this will be the same for all of the kernels. Thus we will prove that

$$\text{tiled}(i, k) = \text{TTMQR}(i, \text{piv}(i, k), k, j) \quad \text{for } j > k.$$

Note that j represents the column in which the updates are applied and all columns j for $j > k$ have the same update history. In Algorithm 3.3, the two j -loops spawn

mutually independent tasks. Since we have an unbounded number of processors, these tasks can all run simultaneously. So j represents any one of these columns.

We will proceed by induction on k . For the first column, $k = 1$, we do not have any dependencies which concern the *GEQRT* operations. Thus from (3.2b) we have for $1 \leq i \leq p$,

$$GEQRT(i, 1) = 4 \quad (3.4)$$

$$UNMQR(i, 1, j) = 4 + 6 = 10 \quad (3.5)$$

Since each column in the coarse-grain elimination list is composed of one or more time steps, we must also proceed with induction on the time steps. Let

$$s_1 = \min_{2 \leq i \leq p} (coarse(i, 1)). \quad (3.6)$$

In the case $k = 1$, we have that

$$s_1 = 1.$$

In other words, the first tasks finish at time step 1 for the coarse-grain algorithm. This is a complicated manner in which to state that $s_1 = 1$, but it will be needed in the general setting.

So for s_1 , from (3.2c) and (3.2d) we have

$$\begin{aligned} TTQRT(I_{s_1}, piv(I_{s_1}, 1), 1) &= \max \begin{pmatrix} GEQRT(piv(I_{s_1}, 1), 1) \\ GEQRT(I_{s_1}, 1) \end{pmatrix} + 2 \\ &= 4 + 2 \end{aligned}$$

thus

$$TTQRT(I_{s_1}, piv(I_{s_1}, 1), 1) = 4 + 2s_1. \quad (3.7)$$

Now from (3.2e), (3.2f), and (3.2g) we have

$$\begin{aligned}
TTMQR(I_{s_1}, \text{piv}(I_{s_1}, 1), 1, j) &= \max \begin{pmatrix} TTQRT(I_{s_1}, \text{piv}(I_{s_1}, 1), 1) \\ UNMQR(\text{piv}(I_{s_1}, 1), 1, j) \\ UNMQR(I_{s_1}, 1, j) \end{pmatrix} + 6 \\
&= 10 \cdot 1 + 6s_1 \\
&= 10 \cdot 1 + 6 \cdot \text{coarse}(I_{s_1}, 1)
\end{aligned}$$

Therefore,

$$TTMQR(I_{s_1}, \text{piv}(I_{s_1}, 1), 1, j) = \text{tiled}(I_{s_1}, 1). \quad (3.8)$$

Assume that for $1 \leq t \leq s-1$ we have

$$TTQRT(I_t, \text{piv}(I_t, 1), 1) = 4 + 2t \quad (3.9)$$

$$TTMQR(I_t, \text{piv}(I_t, 1), 1, j) = 10 + 6t \quad (3.10)$$

then from (3.2c), (3.2d), and (3.3a) we have

$$\begin{aligned}
TTQRT(I_s, \text{piv}(I_s, 1), 1) &= \max \begin{pmatrix} GEQRT(\text{piv}(I_s, 1), 1) \\ GEQRT(I_s, 1) \\ TTQRT(I_{s-1}, \text{piv}(I_{s-1}, 1), 1) \end{pmatrix} + 2 \\
&= \max \begin{pmatrix} 4 \\ 4 \\ 4 + 2(s-1) \end{pmatrix} + 2
\end{aligned}$$

Thus

$$TTQRT(I_s, \text{piv}(I_s, 1), 1) = 4 + 2s. \quad (3.11)$$

From (3.2e), (3.2f), (3.2g), and (3.3b) we have

$$\begin{aligned}
TTMQR(I_s, \text{piv}(I_s, 1), 1, j) &= \max \left(\begin{array}{c} TTQRT(I_s, \text{piv}(I_s, 1), 1) \\ UNMQR(\text{piv}(I_s, 1), 1, j) \\ UNMQR(I_s, 1, j) \\ TTMQR(I_{s-1}, \text{piv}(I_{s-1}, 1), 1, j) \end{array} \right) + 6 \\
&= \max \left(\begin{array}{c} 4 + 2s \\ 10 \\ 10 \\ 10 + 6(s - 1) \end{array} \right) + 6 \\
&= 10 + 6(s - 1) + 6 \\
&= 10 + 6s \\
&= 10 \cdot 1 + 6 \cdot \text{coarse}(I_s, 1)
\end{aligned}$$

Thus

$$TTMQR(I_s, \text{piv}(I_s, 1), 1, j) = \text{tiled}(I_s, 1) \quad (3.12)$$

establishing our base case for the induction on k .

Now assume that for $1 \leq h \leq k - 1$ we have, for any s in column h ,

$$TTMQR(I_s, \text{piv}(I_s, h), h, j) = \text{tiled}(I_s, h).$$

In order to start the elimination of the next column, we must have that all updates from the elimination of the previous column are complete. Thus using the induction assumption, we have

$$GEQRT(i, k) = TTMQR(i, \text{piv}(i, k - 1), k - 1, k) + 4$$

so that

$$GEQRT(i, k) = 10(k - 1) + 6 \cdot \text{coarse}(i, k - 1) + 4 \quad (3.13)$$

and

$$UNMQR(i, k, j) = \max \left(\begin{array}{c} GEQRT(i, k) \\ TTMQR(i, piv(i, k-1), k-1, j) \end{array} \right) + 6$$

so that

$$UNMQR(i, k, j) = 10k + 6 \cdot coarse(i, k-1). \quad (3.14)$$

Again, we must proceed with an induction on the time steps in column k . Let

$$s_1 = \min_{k+1 \leq i \leq p} (coarse(i, k)). \quad (3.15)$$

From (3.2c) and (3.2d) we have

$$\begin{aligned} TTQRT(I_{s_1}, piv(I_{s_1}, k), k) &= \max \left(\begin{array}{c} GEQRT(piv(I_{s_1}, k), k) \\ GEQRT(I_{s_1}, k) \end{array} \right) + 2 \\ &= \max \left(\begin{array}{c} 10(k-1) + 6 \cdot coarse(piv(I_{s_1}, k), k-1) + 4 \\ 10(k-1) + 6 \cdot coarse(I_{s_1}, k-1) + 4 \end{array} \right) + 2 \\ &= 10(k-1) + 6 \max \left(\begin{array}{c} coarse(piv(I_{s_1}, k), k-1) \\ coarse(I_{s_1}, k-1) \end{array} \right) + 4 + 2 \end{aligned}$$

From the application of Lemma 3.4, we have

$$coarse(I_{s_1}, k) - 1 = \max \left(\begin{array}{c} coarse(I_{s_1, k}, k-1) \\ coarse(piv(I_{s_1, k}, k), k-1) \end{array} \right)$$

such that

$$TTQRT(I_{s_1}, piv(I_{s_1}, k), k) = 10(k-1) + 6 [coarse(I_{s_1}, k) - 1] + 4 + 2.$$

Therefore,

$$TTQRT(I_{s_1}, piv(I_{s_1}, k), k) = 10(k-1) + 6s_1. \quad (3.16)$$

For the updates, we must again examine the three dependencies which result from (3.2e), (3.2f), and (3.2g) such that we have

$$\begin{aligned}
TTMQR(I_{s_1}, \text{piv}(I_{s_1}, k), k, j) &= \max \begin{pmatrix} UNMQR(\text{piv}(I_{s_1}, k), k, j) \\ UNMQR(I_{s_1}, k, j) \\ TTQRT(I_{s_1}, \text{piv}(I_{s_1}, k), k) \end{pmatrix} + 6 \\
&= \max \begin{pmatrix} 10k + 6 \cdot \text{coarse}(I_{s_1}, k - 1) \\ 10k + 6 \cdot \text{coarse}(\text{piv}(I_{s_1}, k), k - 1) \\ 10(k - 1) + 6s_1 \end{pmatrix} + 6
\end{aligned}$$

Using Lemma 3.4, we have

$$\begin{aligned}
TTMQR(I_{s_1}, \text{piv}(I_{s_1}, k), k, j) &= \max \begin{pmatrix} 10k + 6(s_1 - 1) \\ 10(k - 1) + 6s_1 \end{pmatrix} + 6 \\
&= 10k + \max \begin{pmatrix} 6s_1 - 6 \\ 6s_1 - 10 \end{pmatrix} + 6 \\
&= 10k + 6s_1
\end{aligned}$$

Therefore

$$TTMQR(I_{s_1}, \text{piv}(I_{s_1}, k), k, j) = \text{tiled}(I_{s_1}, k). \quad (3.17)$$

Now assume that for $s_1 \leq t \leq s - 1$ we have

$$TTQRT(I_t, \text{piv}(I_t, k), k) \leq 10(k - 1) + 6t \quad (3.18)$$

$$TTMQR(I_t, \text{piv}(I_t, k), k, j) = 10k + 6t. \quad (3.19)$$

and note that we do not have equality for $s > s_1$ via Lemma 3.4. From (3.2c), (3.2d), and (3.3a) we have

$$\begin{aligned} TTQRT(I_s, \text{piv}(I_s, k), k) &= \max \begin{pmatrix} GEQRT(\text{piv}(I_s, k), k) \\ GEQRT(I_s, k) \\ TTQRT(I_{s-1}, \text{piv}(I_{s-1}, k), k) \end{pmatrix} + 2 \\ &\leq \max \begin{pmatrix} 10(k-1) + 6 \cdot \text{coarse}(\text{piv}(I_s, k), k-1) + 4 \\ 10(k-1) + 6 \cdot \text{coarse}(I_s, k-1) + 4 \\ 10(k-1) + 6(s-1) \end{pmatrix} + 2 \end{aligned}$$

Note that from Lemma 3.4

$$s-1 \geq \max \begin{pmatrix} \text{coarse}(\text{piv}(I_s, k), k-1) \\ \text{coarse}(I_s, k-1) \end{pmatrix}$$

such that

$$TTQRT(I_s, \text{piv}(I_s, k), k) \leq 10(k-1) + 6(s-1) + 4 + 2.$$

Thus

$$TTQRT(I_s, \text{piv}(I_s, k), k) \leq 10(k-1) + 6s. \quad (3.20)$$

For the updates, we must examine the four dependencies which result from (3.2e), (3.2f), (3.2g), and (3.3b) such that we have

$$\begin{aligned} TTMQR(I_s, \text{piv}(I_s, k), k, j) &= \max \begin{pmatrix} UNMQR(\text{piv}(I_s, k), k, j) \\ UNMQR(I_s, k, j) \\ TTQRT(I_s, \text{piv}(I_s, k), k) \\ TTMQR(I_{s-1}, \text{piv}(I_{s-1}, k), k, j) \end{pmatrix} + 6 \\ &= \max \begin{pmatrix} 10k + 6 \cdot \text{coarse}(I_s, k-1) \\ 10k + 6 \cdot \text{coarse}(\text{piv}(I_s, k), k-1) \\ 10(k-1) + 6s \\ 10k + 6(s-1) \end{pmatrix} + 6. \end{aligned}$$

As before, Lemma 3.4 allows us to write

$$\begin{aligned}
TTMQR(I_s, \text{piv}(I_s, k), k, j) &= \max \begin{pmatrix} 10k + 6(s-1) \\ 10(k-1) + 6s \end{pmatrix} + 6 \\
&= 10k + \max \begin{pmatrix} 6s - 6 \\ 6s - 10 \end{pmatrix} + 6 \\
&= 10k + 6s
\end{aligned}$$

Therefore

$$TTMQR(I_s, \text{piv}(I_s, k), k, j) = \text{tiled}(i, k). \quad (3.21)$$

■

Corollary 3.6 *Given an elimination list for a coarse-grain algorithm on a matrix of size $p \times q$ where $p > q$, the critical path length of the corresponding tiled algorithm is bounded by*

$$\text{tiled}(p, q-1) + 4 + 2 \leq CP(p, q) < \text{tiled}(p, q).$$

Proof: For any tiled matrix, the last column will necessarily need to be factorized which explains the addition of four time steps and since $p > q$ at least one $TTQRT$ will be present which accounts for the two time steps thereby establishing the lower bound. By including one more column, the upper bound not only includes the factorization of column q , but also the respective updates onto column $q+1$ such that the critical path of the $p \times q$ tiled matrix must be smaller. ■

Corollary 3.7 *Given an elimination list for a coarse-grain algorithm on a matrix of size $p \times q$ where $p = q$, the critical path length of the corresponding tiled algorithm is*

$$CP(p, q) = \text{tiled}(p, q-1) + 4.$$

Proof: In the last column, we need only to factorize the diagonal tile which explains the additional four time steps. Moreover, there are no further columns to apply any

updates to nor any tiles below the diagonal that need to be eliminated. Thus the result is obtained. \blacksquare

In the remainder of this chapter, we will make use of diagrams to clarify certain aspects of the proofs and provide examples to further illustrate the points being made. These diagrams make use of the kernel representations as shown in Figure 3.1.





kernel		weight	kernel		weight
GEQRT		4	UNMQR		6
TTQRT		2	TTMQR		6

Figure 3.1: Icon representations of the kernels

We have a closed-form expression for the critical path of tiled FLATTREE for all three cases: single tiled column, square tiled matrix, and rectangular tiled matrix of more than one column.

Proposition 3.8 *Consider a tiled matrix of size $p \times q$, where $p \geq q \geq 1$. The critical path length of FLATTREE is*

$$CP_{ft}(p, q) = \begin{cases} 2p + 2, & \text{if } q = 1; \\ 22p - 24, & \text{if } p = q > 1; \\ 6p + 16q - 22, & \text{if } p > q > 1. \end{cases}$$

Proof: Consider first the case $q = 1$. We shall proceed by induction on p to show that the critical path of FLATTREE is of length $2p + 2$. If $p = 1$, then from Table 3.1 the result is obtained since only $GEQRT(1, 1)$ is required. With the base case established, now assume that this holds for all $p - 1 > q = 1$. Thus at time $t = 2(p - 1) + 2 = 2p$, we have that for all $p - 1 \geq i \geq 1$ tile $(i, 1)$ has been factorized into a triangle and for all $p - 1 \geq i > 1$, tile $(i, 1)$ has been zeroed out. Therefore, tile $(p, 1)$ will be zeroed out with $TTQRT(p, 1)$ at time $t + 2 = 2(p - 1) + 2 + 2 = 2p + 2$.

Considering the second case $p = q > 1$, we will be using Figure 3.2 to illustrate. We initialize with a triangularization of the first column and send the update to the remaining column(s), 10 time units. Then we fill the pipeline with the updates onto the remaining column(s) from the zeroing operations of the first column, $6(p-1)$ time units. Then for each column after the first, except the last one, we fill the pipeline with the triangularization, update of triangularization, and update of zeroing for the bottom most tile, $(4+6+6)(p-2)$ time units. In the last column, we then triangularize the bottom most tile, 4 time units. Thus

$$10 + 6(p-1) + (4+6+6)(q-2) + 4 = 6p + 16q - 24 = 22p - 24$$

The third case is analogous to the second case but we still need to zero out the bottom most tile in the last column which explains the difference of 2 in the formula from the square case. ■

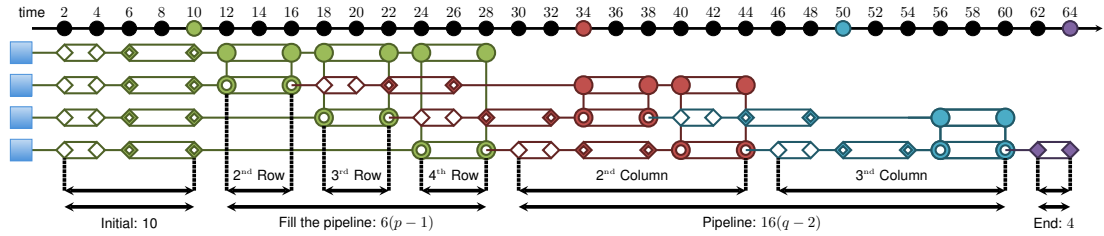


Figure 3.2: Critical Path length for the weighted FLATTREE on a matrix of 4×4 tiles.

We remind that for the coarse algorithm,

$$coarse(p, q) = \begin{cases} 0, & \text{if } q < 1; \\ 0, & \text{if } p = q = 1; \\ p + q - 2, & \text{if } p > q \geq 1; \\ 2p - 3, & \text{if } p = q > 1. \end{cases}$$

So we find that considering a tiled matrix of size $p \times q$, where $p \geq q \geq 1$. The critical path length of FLATTREE is given as

$$CP(p, q) = 10(q-1) + 6(coarse(p, q-1)) + 4 + 2(coarse(p, q) - coarse(p, q-1)).$$

Proposition 3.9 *The critical path length of FIBONACCI is greater than $22q - 30$ and less than $22q + 6\lceil\sqrt{2p}\rceil$.*

Proof: The critical path length of the coarse-grain FIBONACCI algorithm for a $p \times q$ matrix is

$$\text{coarse}(p, q) = x + 2q - 2.$$

Thus from Proposition 3.6 we have

$$10(q - 1) + 6(x + 2(q - 1) - 2) + 4 \leq CP(p, q) < 10q + 6(x + 2q - 2).$$

Recall that x is the least integer such that

$$\frac{x(x + 1)}{2} \geq p - 1$$

whereby

$$x = -\frac{1}{2} + \frac{\sqrt{8p - 7}}{2}.$$

Thus $x \leq \lceil\sqrt{2p}\rceil$ and therefore

$$22q - 30 < CP(p, q) < 22q - 12 + 6\lceil\sqrt{2p}\rceil.$$

■

Similarly to [25] in which an iterate of a column is defined for the coarse-grain algorithms, we define a weighted iterate and our notation will follow in the same manner.

A column of length n is a sequence of n integers:

$$a = a_1^{n_1} \cdots a_q^{n_q}$$

where power means concatenation with the following restrictions:

$$a_1 \geq 0 \quad a_{i+1} > a_i, \quad 1 \leq i \leq q - 1;$$

$$n_i > 0, \quad 1 \leq i \leq q; \quad n_1 + \cdots + n_q = n.$$

We define on the set of columns of length n the classical partial ordering of \mathbb{R}^n :

$$x \leq y \iff (x_i \leq y_i, i \leq i \leq n)$$

and the s -truncate ($1 \leq s \leq n$) of a is a column of length s composed of the s first elements of a and is denoted a^s .

Definition 3.10 *Given a task weight of w and column $a = a_1^{n_1} \cdots a_q^{n_q}$, the column $c = c_1^{m_1} \cdots c_p^{m_p}$ is called an iterate of a , or $c = \text{iter}(a)$, if*

(i) c is a column of length $n - 1$

(ii) $a_1 + w \leq c_1$

(a) if $a_1 + w \leq c_1 \leq a_2$ then $m_1 \leq \lfloor n_1/2 \rfloor$

(b) if there exists an h such that $a_{k-1} + w \leq c_h \leq a_k$ then

$$m_h \leq \lfloor (n_1 + \cdots + n_{k-1} - m_1 - \cdots - m_{h-1}) / 2 \rfloor$$

for $2 \leq k \leq q$ and $1 \leq h \leq p$ with $m_0 = 0$.

(c) else $a_j \leq c_h$ and

$$m_h \leq \lfloor (n_1 + \cdots + n_j - m_1 - \cdots - m_{h-1}) / 2 \rfloor$$

where $j = \min(p + 1, q)$.

Definition 3.11 *Given a task weight of w , let $a = a_1^{n_1} \cdots a_q^{n_q}$ be a column iterate of length n then the sequence $b = b_1^{m_1} \cdots b_p^{m_p}$, or $b = \text{opiter}(a)$, is defined as*

(i) for b_1 and m_1

(a) if $n_1 = 1$, then $b_1 = a_2 + w$ and $m_1 = \lfloor (n_1 + n_2) / 2 \rfloor$.

(b) if $n_1 > 1$, then $b_1 = a_1 + w$ and $m_1 = \lfloor (n_1) / 2 \rfloor$.

(ii) if there exists k such that $a_{k-1} + w \leq b_{i-1} \leq a_k$, then

$$r_{i-1} = n_1 + \cdots + n_{k-1} - m_1 - \cdots - m_{i-1} \geq 1.$$

(a) if $b_{i-1} < a_k$ and $r_{i-1} > 1$, then $b_i = b_{i-1} + w$, and $m_i = \lfloor r_{i-1}/2 \rfloor$.

(b) else $b_i = a_k + w$, $m_i = \lfloor (n_k + r_{i-1})/2 \rfloor$.

(iii) if $b_{i-1} > a_j$ where $j = \min(i, q)$, then $b_i = b_{i-1} + w$, and

$$m_i \leq \lfloor (n_1 + \cdots + n_j - m_1 - \cdots - m_{i-1})/2 \rfloor.$$

Proposition 3.12 *Given an iterated column a of length n , the sequence*

$$b = b_1^{m_1} \cdots b_p^{m_p},$$

or $b = \text{optiter}(a)$ is an iterate of a .

Proof: The proof follows directly from the definition. ■

Proposition 3.13

(i) *Let a_n be a column of length n and $c_{n-1} = \text{iter}(a_n)$ an iterated column of a_n .*

Then

$$b_{n-1} = \text{optiter}(a_n) \leq \text{iter}(a_n) = c_{n-1}.$$

(ii) *Let a_n and c_n be two columns of length n such that $a_n \leq c_n$. Then*

$$\text{optiter}(a_n) \leq \text{optiter}(c_n).$$

Proof:

(i) Clearly, $b_1 \leq c_1$ by definition since b_1 is chosen to be as small as possible.

Moreover, by definition $b_i \leq c_j$ for $i \leq j$ since (i) if $b_{i-1} < a_k$ and $r_{i-1} > 1$, meaning there are enough elements available to perform a pairing, then b_i is

again chosen as small as possible, (ii) otherwise $b_i = a_k + w$ which is the smallest again, (iii) else $b_{i-1} > a_j$ and b_i is chosen as the next smallest element. Thus

$$b_{n-1}^{m_1+\dots+m_i} \leq c_{n-1}^{m_1+\dots+m_i}, \quad 1 \leq i \leq p$$

so that $b_{n-1} \leq c_{n-1}$.

- (ii) This is another direct application of the definition and follows along the same argument.

■

Clearly, letting $w = 1$ gives the definitions of *iter* and *optiter* of the coarse-grain algorithms as presented in [25]. Definition 3.11 is the ASAP algorithm on a single tiled column and can be viewed as the counter part of the coarse-grain GREEDY algorithm in the tiled case and follows a bottom to top elimination of the tiles. In order to preserve the bottom to top elimination, the weight of the updates must be an integer multiple of the iterated column weight.

Theorem 3.14 *Given a matrix of $p \times q$ tiles, a factorization kernel weight of γ , an elimination kernel weight of α , and an update kernel weight of $\beta = n\alpha$ for some $n \in \mathbb{N}$, the GRASAP algorithm is optimal in the context of the class of algorithms that progress left to right, bottom to top.*

Proof: From Theorem 3.5 we have a direct translation from any coarse-grain algorithm in this class to the tiled algorithm for the first $q - 1$ columns. Thus we are given the time steps at which rows in column q are available for elimination. Now we fix the time steps for the elimination of the last column and follow whatever tree the algorithm provides for this last column. We can replace the elimination of the first $q - 1$ columns and updates from these eliminations onto the remaining columns with the tiled GREEDY algorithm. This is possible since the translation function is monotonically increasing and we know that GREEDY is optimal for the coarse-grain

algorithms and therefore optimal for the first $q - 1$ columns in the tiled algorithms. In another manner of speaking, we slow down the eliminations and updates on the first $q - 1$ columns when not using the tiled GREEDY algorithm. (An illustrative example is shown in Figure 3.3b.)

Let c be the next to last column of the coarse-grain elimination table which is of length $p - (q - 1) + 1$. Now letting

$$a = (\gamma + \beta)(q - 1) + \beta \cdot \text{coarse}(p, q - 1) + \gamma$$

provides an iterated column of length $p - (q - 1) + 1$ for the tiled algorithm. With $w = \alpha$ we have that $b = \text{optiter}(a)$ is an optimal iterated column of length $p - q + 1$ with the elimination progressing from bottom to top. This can be applied to any tiled algorithm in this class since we only concern ourselves with the time steps at which the last column's elements are available for elimination. In other words, this is a speeding up of the elimination of the last column while adhering to any restrictions incurred from the previous columns. (An illustrative example is shown in Figure 3.3c.)

Combining these two ideas, let GREEDY be performed on the first $q - 1$ columns and then ASAP on the remaining column q . This provides an optimal algorithm in this class of algorithms. ■

In Figure 3.4 we provide an illustrated example of the GREEDY and GRASAP algorithms on a matrix of 15×2 tiles where the operations are given by Figure 3.1.

It can be seen that GRASAP finishes before GREEDY since GREEDY must wait and progress with the same elimination scheme as the coarse-grain algorithm while GRASAP can begin eliminating in the last column as soon as a pair of tiles becomes available. (The elimination of the first column is shown in light gray.)

We have analyzed the critical path length of GRASAP versus that of GREEDY for tiled matrices $p \times q$ where $1 \leq p \leq 100$ and $1 \leq q \leq p$ (see Figure 3.5). In all cases where there is a difference (which is just over 44% of the cases), the difference is always two time steps.

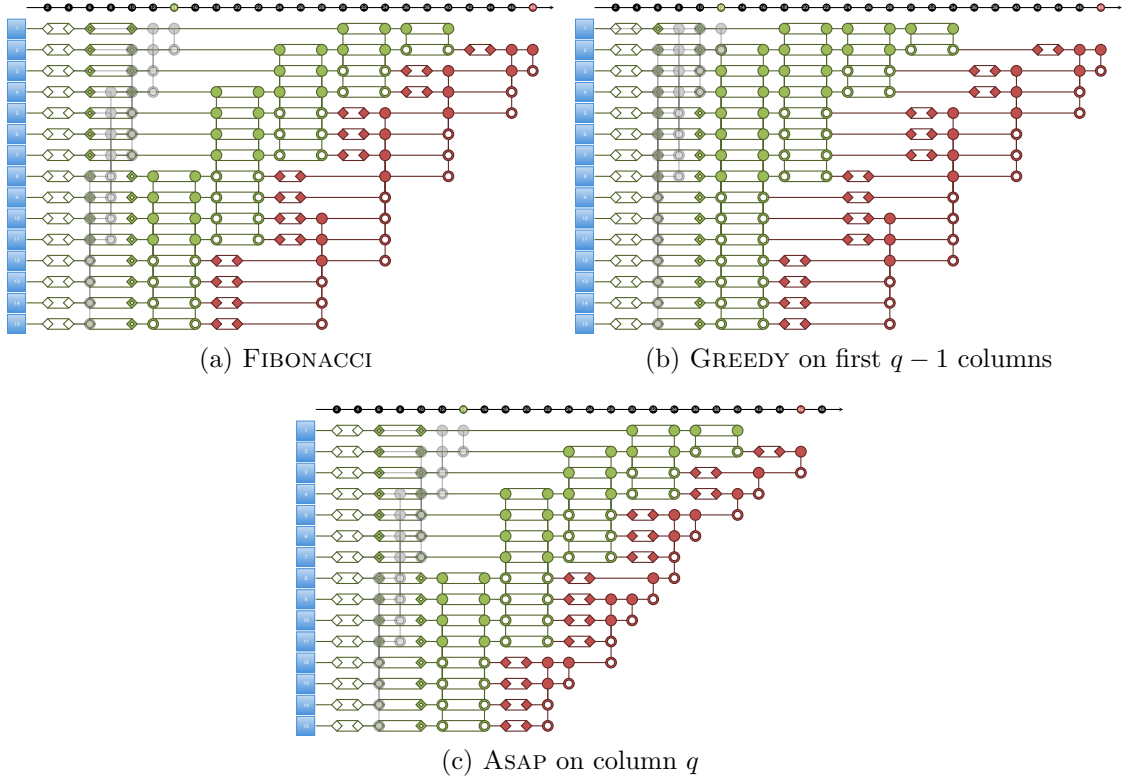


Figure 3.3: Illustration of first and second parts of the proof of Theorem 3.14 using the FIBONACCI algorithm on a matrix of 15×2 tiles.

We now show that without having the update kernel weight an integer multiple of the elimination kernel weight, the bottom to top progression is nullified and we cannot provide optimality of the algorithm.

Assume that the update kernel weight is 3 and the elimination kernel weight is 2. Let $a_{11} = 3^7 6^4$ be column from some elimination scheme. We shall apply three iterated schemes to this column: (1) an ASAP scheme that progresses from bottom to top, (2) an ASAP scheme that can progress in any manner, and (3) an ASAP scheme which may provide a lag.

In Table 3.5 we clearly see that elimination scheme (3) provides the best time for the algorithm. The reason is that enough of a lag was provided such that a binomial tree could progress without hindrance. Therefore without integer multiple weights on the update kernel, we cannot know which scheme will be optimal.

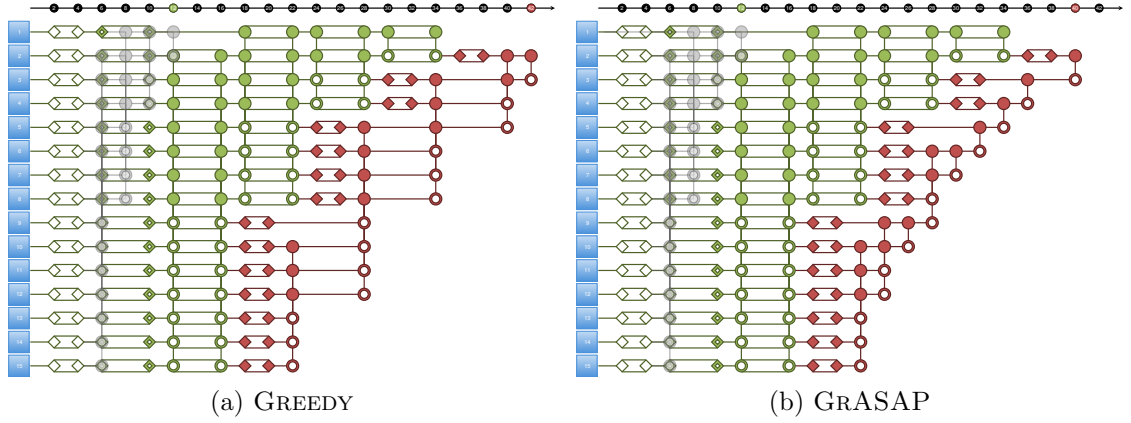


Figure 3.4: GREEDY versus GRASAP on matrix of 15×2 tiles.

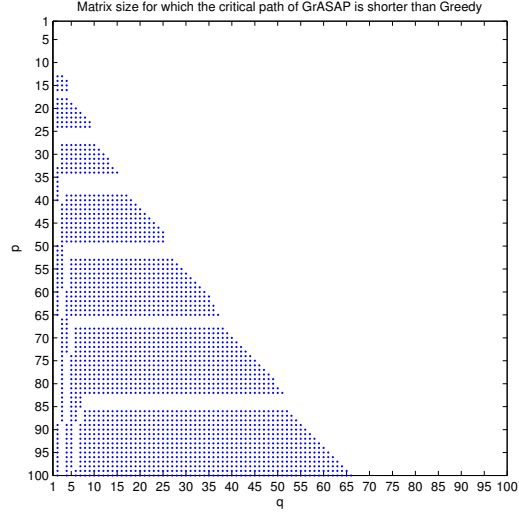


Figure 3.5: Tiled matrices of $p \times q$ where the critical path length of GRASAP is shorter than that of GREEDY for $1 \leq p \leq 100$ and $1 \leq q \leq p$.

The PLASMA library provides more algorithms, that can be informally described as trade-offs between FLATTREE and BINARYTREE. (We remind that FLATTREE is the same as algorithm as SAMEH-KUCK.) These algorithms are referred to as PLASMATREE in all the following, and differ by the value of an input parameter called the *domain size* BS . This domain size can be any value between 1 and p , inclusive. Within a domain, that includes BS consecutive rows, the algorithm works just as FLATTREE: the first row of each domain acts as a local panel and is used to zero out the tiles in all the other rows of the domain. Then the domains are merged:

the panel rows are zeroed out by a binary tree reduction, just as in BINARYTREE. As the algorithm progresses through the columns, the domain on the very bottom is reduced accordingly, until such time that there is one less domain. For the case that $BS = 1$, PLASMATREE follows a binary tree on the entire column, and for $BS = p$, the algorithm executes a flat tree on the entire column. It seems very difficult for a user to select the domain size BS leading to best performance, but it is known that BS should increase as q increases. Table 3.3 shows the time-steps of PLASMATREE with a domain size of $BS = 5$. In the experiments of §3.3, we use all possible values of BS and retain the one leading to the best value.

3.3 Experimental results

All experiments were performed on a 48-core machine composed of eight hexa-core AMD Opteron 8439 SE (codename Istanbul) processors running at 2.8 GHz. Each core has a theoretical peak of 11.2 Gflop/s with a peak of 537.6 Gflop/s for the whole machine. The Istanbul micro-architecture is a NUMA architecture where each socket has 6 MB of level-3 cache and each processor has a 512 KB level-2 cache and a 128 KB level-1 cache. After having benchmarked the AMD ACML and Intel MKL BLAS libraries, we selected MKL (10.2) since it appeared to be slightly faster in our experimental context. Linux 2.6.32 and Intel Compilers 11.1 were also used in conjunction with PLASMA 2.3.1.

For all results, we show both double and double complex precision, using all 48 cores of the machine. The matrices are of size $m = 8000$ and $200 \leq n \leq 8000$. The tile size is kept constant at $n_b = 200$, so that the matrices can also be viewed as $p \times q$ tiled matrices where $p = 40$ and $1 \leq q \leq 40$. All kernels use an inner blocking parameter of $i_b = 32$.

In double precision, an FMA (“*fused multiply-add*”, $y \leftarrow \alpha x + y$) involves three double precision numbers for two flops, but these two flops can be combined into one FMA and thus completed in one cycle. In double complex precision, the operation

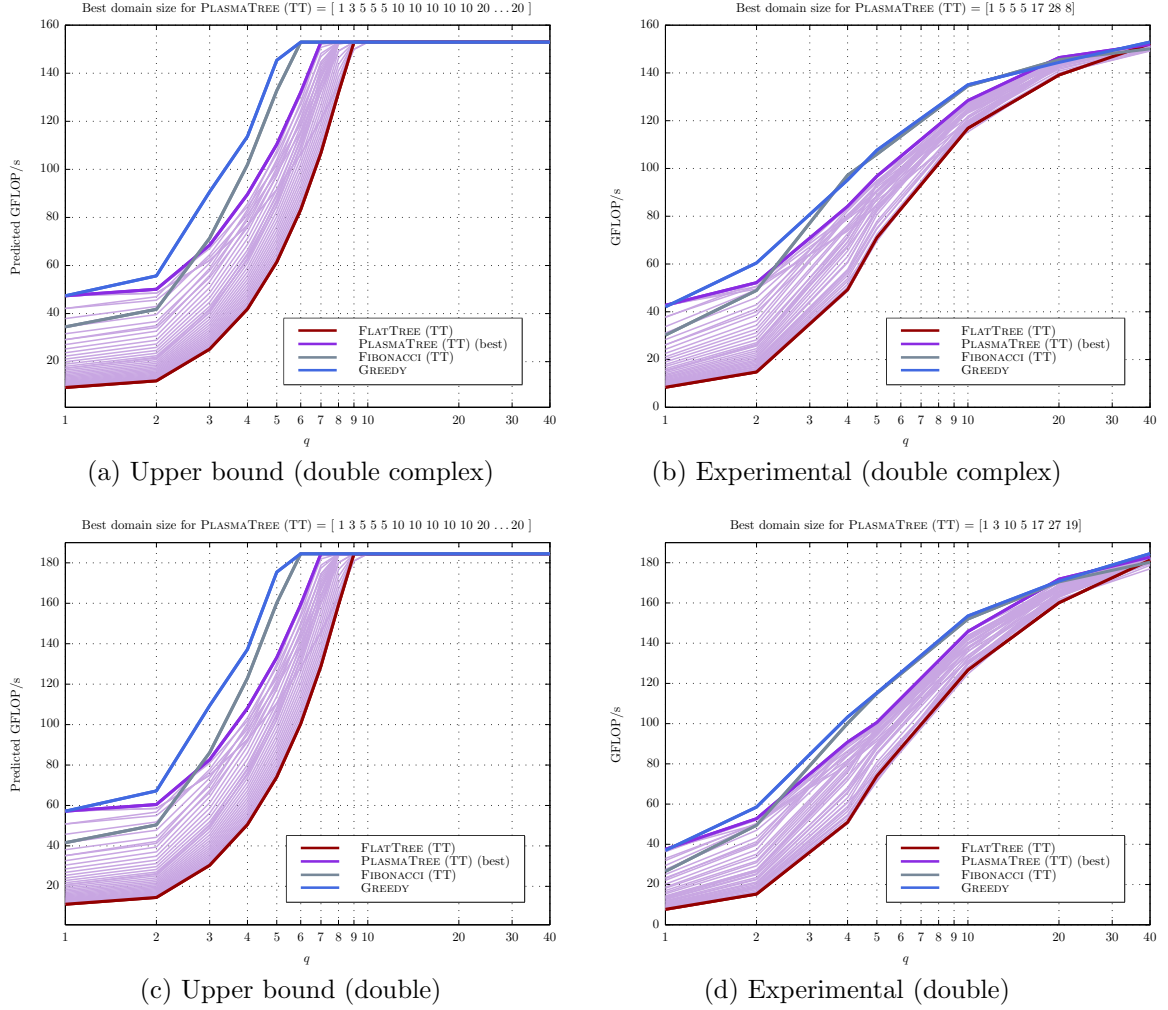
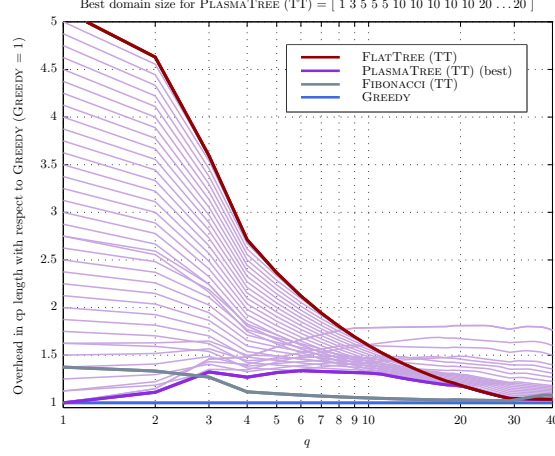


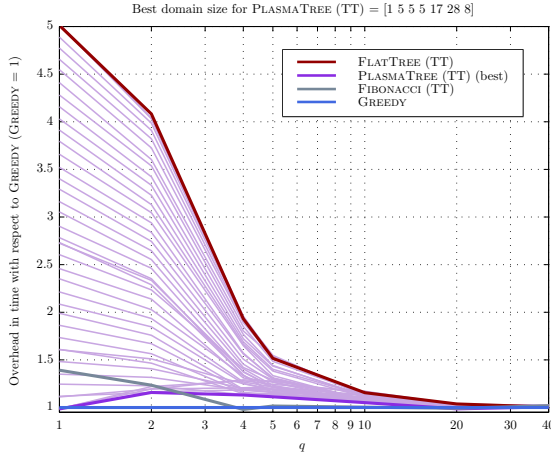
Figure 3.6: Upper bound and experimental performance of QR factorization - TT kernels

$y \leftarrow \alpha x + y$ involves six double precision numbers for eight flops; there is no FMA. The ratio of computation/communication is therefore, potentially, four times higher in double complex precision than in double precision. Communication aware algorithms are much more critical in double precision than in double complex precision.

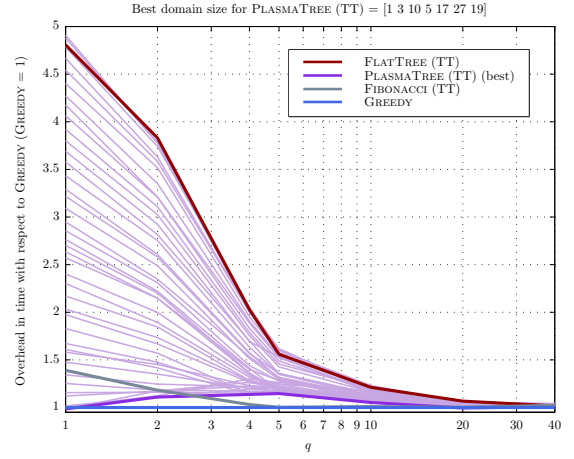
For each experiment, we provide a comparison of the theoretical performance to the actual performance. The theoretical performance is obtained by modeling the limiting factor of the execution time as either the critical path, or the sequential time divided by the number of processors. This is similar in approach to the Roofline



(a) Theoretical CP length



(b) Experimental (double complex)



(c) Experimental (double)

Figure 3.7: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

model [53]. Taking γ_{seq} as the sequential performance, T as the total number of flops, cp as the length of the critical path, and P as the number of processors, the upper bound on performance, γ_{ub} , is

$$\gamma_{ub} = \frac{\gamma_{seq} \cdot T}{\max\left(\frac{T}{P}, cp\right)}$$

Figures 3.6a and 3.6c depict the upper bound on performance of all algorithms which use the *Triangle on top of triangle* kernels. Since PLASMATree provides an additional tuning parameter of the domain size, we show the results for each value of this parameter as well as the composition of the best of these domain sizes. Again, it

is not evident what the domain size should be for the best performance, hence our exhaustive search.

Part of our comprehensive study also involved comparisons made to the Semi-Parallel Tile and Fully-Parallel Tile CAQR algorithms found in [11] which are much the same as those found in PLASMA. As with PLASMA, the tuning parameter BS controls the domain size upon which a flat tree is used to zero out tiles below the root tile within the domain and a binary tree is used to merge these domains. Unlike PLASMA, it is not the bottom domain whose size decreases as the algorithm progresses through the columns, but instead is the top domain. In this study, we found that the PLASMA algorithms performed identically or better than these algorithms and therefore we do not report these comparisons.

Figure 3.6b and 3.6d illustrate the experimental performance reached by GREEDY, FIBONACCI and PLASMATREE algorithms using the TT (*Triangle on top of triangle*) kernels. In both cases, double or double complex precision, the performance of GREEDY is better than PLASMATREE even for the best choice of domain size. Moreover, as expected from the analysis in §3.2.2, GREEDY outperforms FIBONACCI the majority of the time. Furthermore, we see that, for rectangular matrices, the experimental performance in double complex precision matches the upper bound on performance. This is not the case for double precision because communications have higher impact on performance.

While it is apparent that GREEDY does achieve higher levels of performance, the percentage may not be as obvious. To that end, taking GREEDY as the baseline, we present in Figure 3.7 the theoretical, double, and double complex precision overhead for each algorithm that uses the *Triangle on top of triangle* kernel as compared to GREEDY. These overheads are respectively computed in terms of critical path length and time. At a smaller scale (Figure 3.13), it can be seen that GREEDY can perform up to 13.6% better than PLASMATREE.

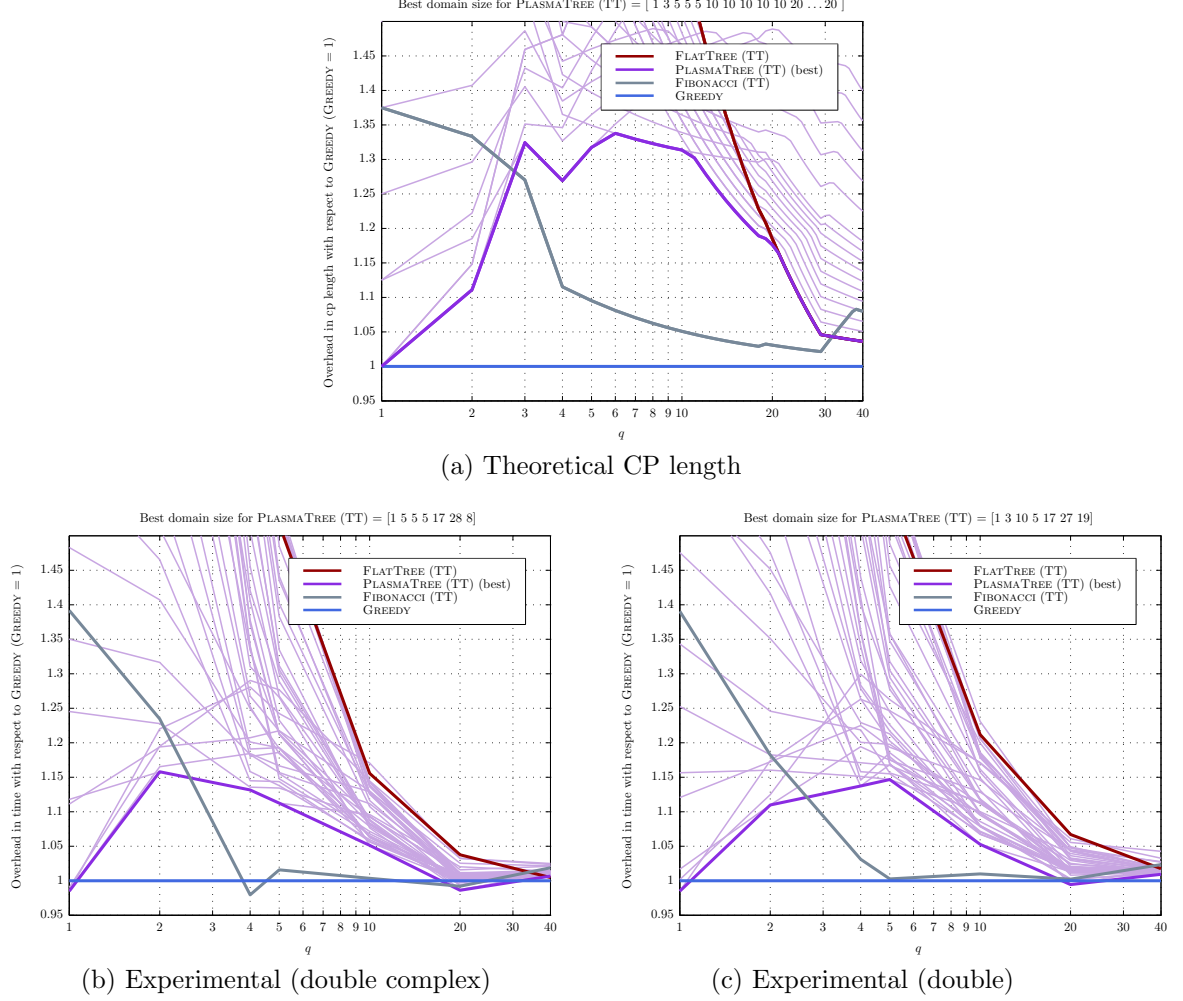


Figure 3.8: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

For all matrix sizes considered, $p = 40$ and $1 \leq q \leq 40$, in the theoretical model, the critical path length for GREEDY is either the same as that of PLASMA TREE ($q = 1$) or is up to 25% shorter than PLASMA TREE ($q = 6$). Analogously, the critical path length for GREEDY is at least 2% to 27% shorter than that of FIBONACCI. In the experiments, the matrix sizes considered were $p = 40$ and $q \in \{1, 2, 4, 5, 10, 20, 40\}$. In double precision, GREEDY has a decrease of at most 1.5% than the best PLASMA TREE ($q = 1$) and a gain of at most 12.8% than the best PLASMA TREE ($q = 5$). In double complex precision, GREEDY has a decrease of at most 1.5% than the best PLASMA TREE ($q = 1$) and a gain of at most 13.6% than the best PLASMA TREE

($q = 2$). Similarly, in double precision, GREEDY provides a gain of 2.6% to 28.1% over FIBONACCI and in double complex precision, GREEDY has a decrease of at most 2.1% and a gain of at most 28.2% over FIBONACCI.

Although it is evidenced that PLASMATREE does not vary too far from GREEDY or FIBONACCI, one must keep in mind that there is a tuning parameter involved and we choose the best of these domain sizes for PLASMATREE to create the composite result, whereas with GREEDY, there is no such parameter to consider. Of particular interest is the fact that GREEDY always performs better than any other algorithm² for $p \gg q$. In the scope of PLASMATREE, a domain size $BS = 1$ will force the use of a binary tree so that both GREEDY and PLASMATREE behave the same. However, as the matrix tends more to a square, i.e., q tends toward p , we observe that the performance of all of the algorithms, including FLATTREE, are on par with GREEDY. As more columns are added, the parallelism of the algorithm is increased and the critical path becomes less of a limiting factor, so that the performance of the kernels is brought to the forefront. Therefore, all of the algorithms are performing similarly since they all share the same kernels.

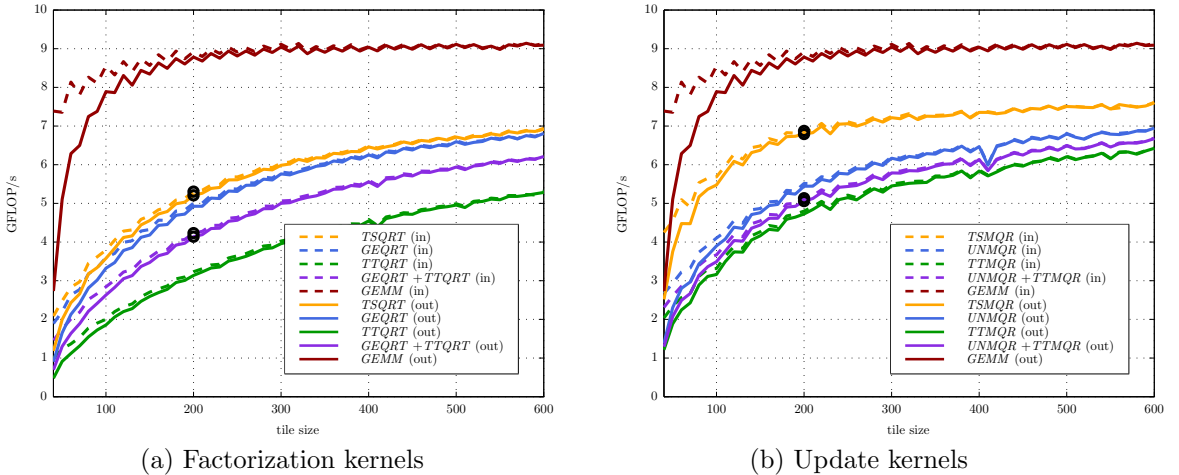


Figure 3.9: Kernel performance for double complex precision

²When $q = 1$, GREEDY and FLATTREE exhibit close performance. They both perform a binary tree reduction, albeit with different row pairings.

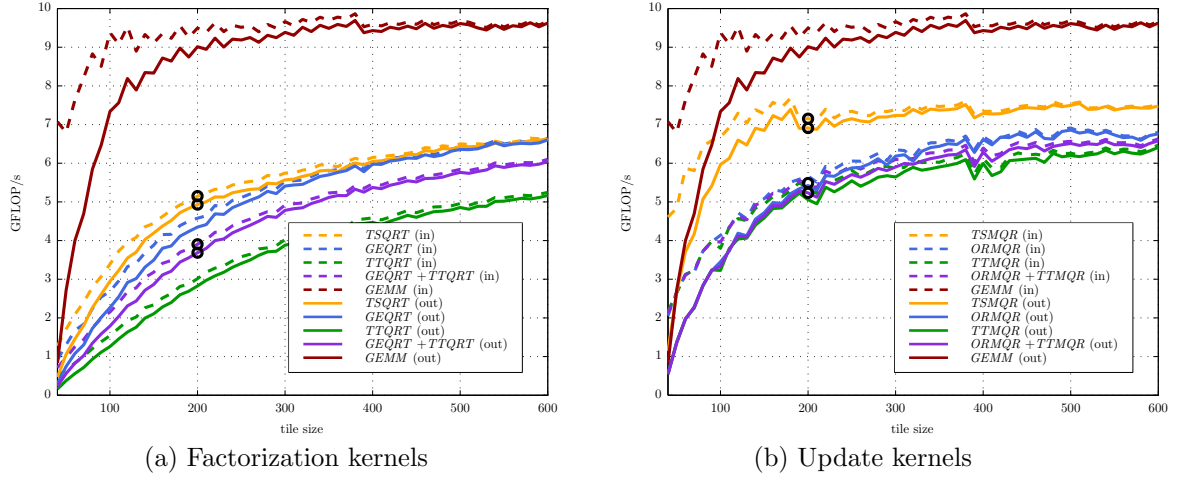


Figure 3.10: Kernel performance for double precision

In order to accurately assess the impact of the kernel selection towards the performance of the algorithms, Figures 3.9 and 3.10 show both the in cache and out of cache performance using the *No Flush* and *MultCallFlushLRU* strategies as presented in [29, 51]. Since an algorithm using *TT* kernels will need to call *GEQRT* as well as *TTQRT* to achieve the same as the *TS* kernel *TSQRT*, the comparison is made between *GEQRT* + *TTQRT* and *TSQRT* (and similarly for the updates). For $n_b = 200$, the observed ratio for in cache kernel speed for *TSQRT* to *GEQRT* + *TTQRT* is 1.3374, and for *TSMQR* to *UNMQR* + *TTMQR* is 1.3207. For out of cache, the ratio for *TSQRT* to *GEQRT* + *TTQRT* is 1.3193 and for *TSMQR* to *UNMQR* + *TTMQR* it is 1.3032. Thus, we can expect about a 30% difference between the selection of the kernels, since we will have instances of using in cache and out of cache throughout the run. Most of this difference is due to the higher efficiency and data locality within the *TT* kernels as compared to the *TS* kernels.

Having seen that kernel performance can have a significant impact, we also compare the *TT* based algorithms to those using the *TS* kernels. The goal is to provide a complete assessment of all currently available algorithms, as shown in Figure 3.11. For double precision, the observed difference in kernel speed is 4.976 GFLOP/sec for

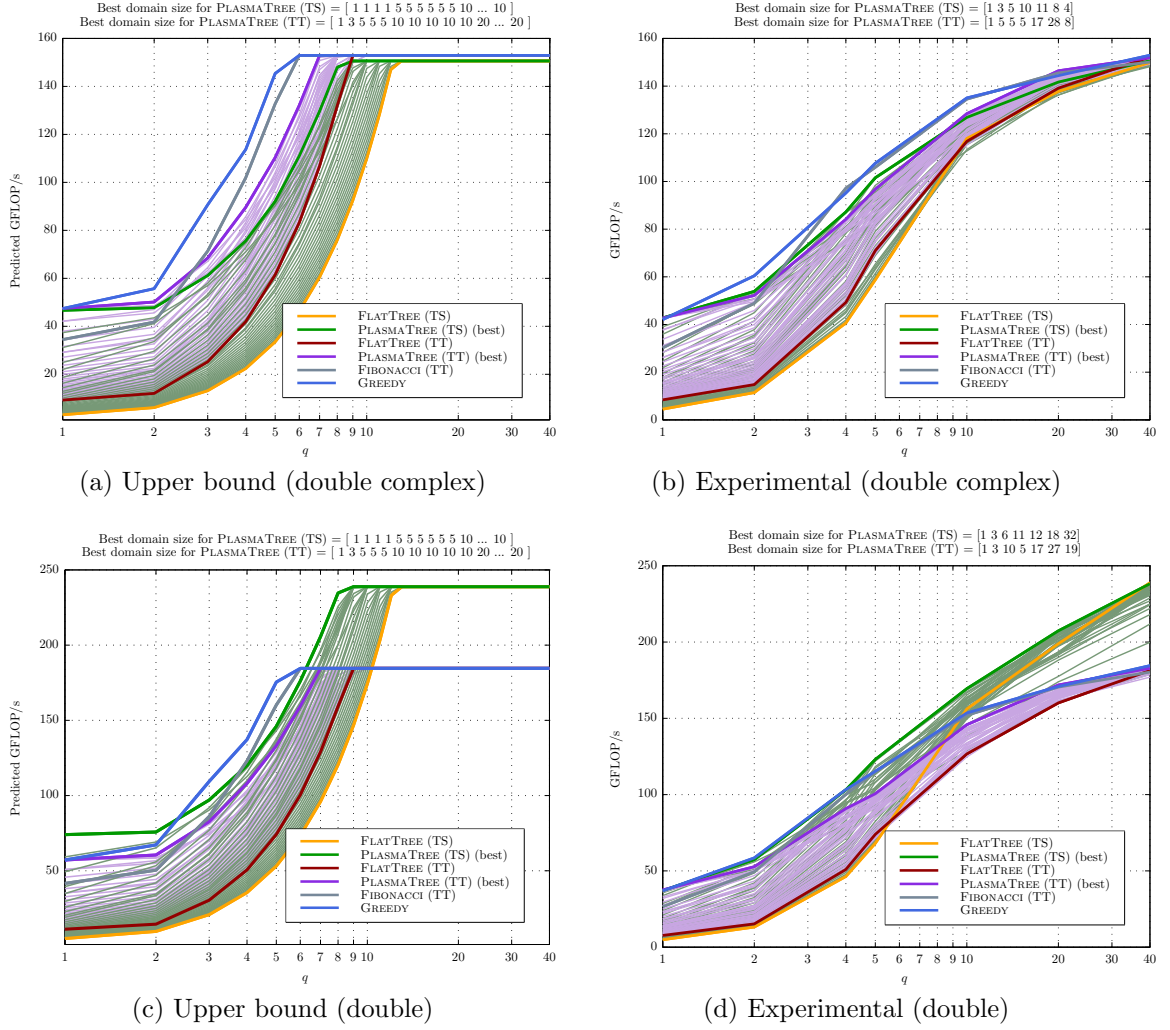
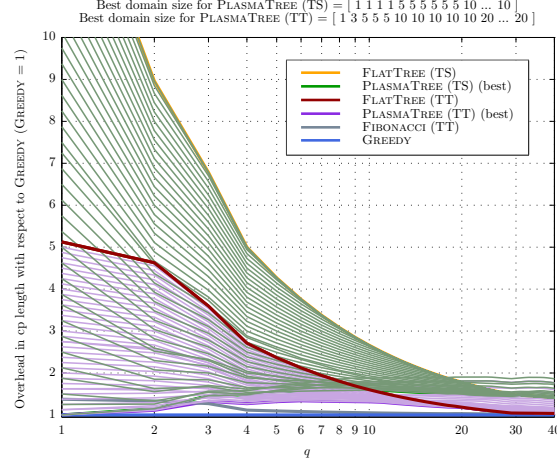
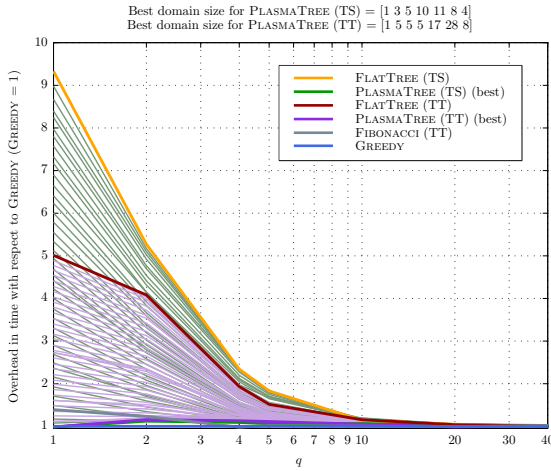


Figure 3.11: Upper bound and experimental performance of QR factorization - All kernels

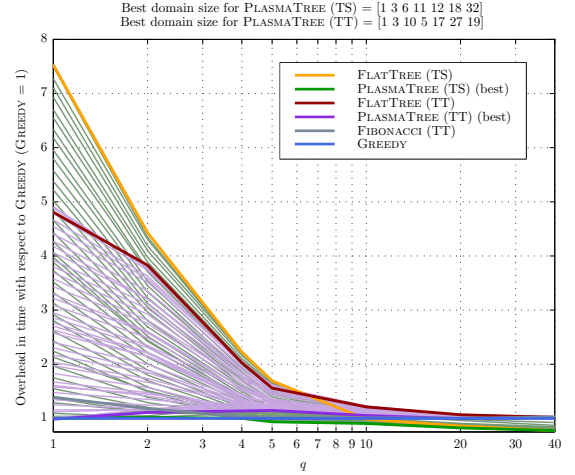
the TS kernels versus 3.844 GFLOP/sec for the TT kernels which provides a ratio of 1.2945 and is in accordance with our previous analysis. It can be seen that as the number of columns increases, whereby the amount of parallelism increases, the effect of the kernel performance outweighs the benefit provided by the extra parallelism afforded through the TT algorithms. Comparatively, in double complex precision, GREEDY does perform better, even against the algorithms using the TS kernels. As before, one must keep in mind that GREEDY does not require the tuning parameter of the domain size to achieve this better performance.



(a) Theoretical CP length



(b) Experimental (double complex)



(c) Experimental (double)

Figure 3.12: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

From these experiments, we showed that in double complex precision, GREEDY demonstrated better performance than any of the other algorithms and moreover, it does so without the need to specify a domain size as opposed to the algorithms in PLASMA. In addition, in double precision, for matrices where $p \gg q$, GREEDY continues to excel over any other algorithm using the TT kernels, and continues to do so as the matrices become more square.

3.4 Conclusion

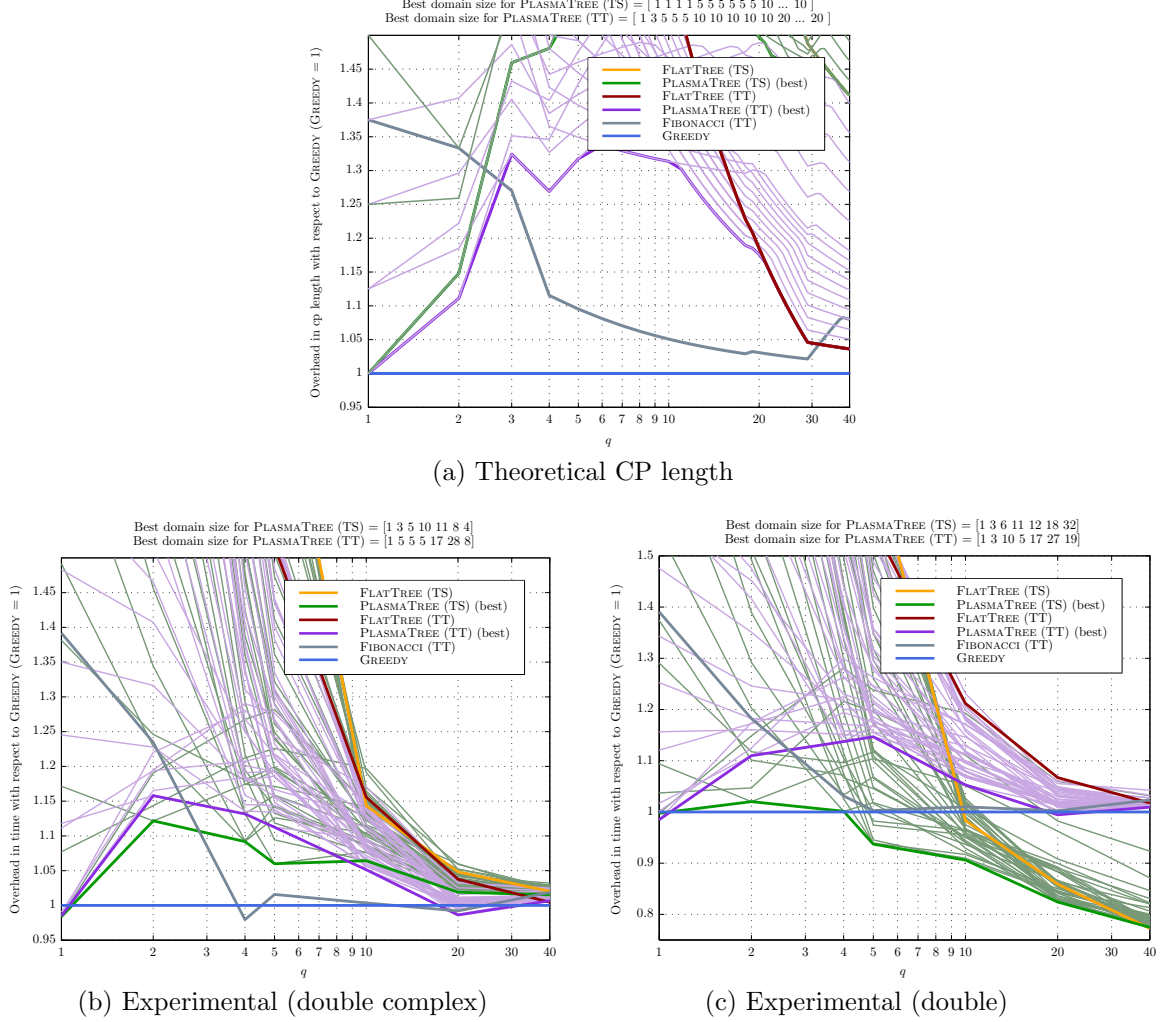


Figure 3.13: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

In this chapter, we have presented FIBONACCI, and GREEDY, two new algorithms for tiled QR factorization. These algorithms exhibit more parallelism than state-of-the-art implementations based on reduction trees. We have provided accurate estimations for the length of their critical path.

Comprehensive experiments on multicore platforms confirm the superiority of the new algorithms for $p \times q$ matrices, as soon as, say, $p \geq 2q$. This holds true when comparing not only with previous algorithms using TT (*Triangle on top of triangle*) kernels, but also with all known algorithms based on TS (*Triangle on top of square*) kernels. Given that TS kernels offer more locality, and benefit from better

elementary arithmetic performance, than TT kernels, the better performance of the new algorithms is even more striking, and further demonstrates that a large degree of a parallelism was not exploited in previously published solutions.

Future work will investigate several promising directions. First, using rectangular tiles instead of square tiles could lead to efficient algorithms, with more locality and still the same potential for parallelism. Second, refining the model to account for communications, and extending it to fully distributed architectures, would lay the ground work for the design of MPI implementations of the new algorithms, unleashing their high level of performance on larger platforms. Finally, the design of robust algorithms, capable of achieving efficient performance despite variations in processor speeds, or even resource failures, is a challenging but crucial task to fully benefit from future platforms with a huge number of cores.

Algorithm 3.4: GREEDY algorithm via TT kernels.

```
1 for  $j = 1$  to  $q$  do
    /*  $nZ(j)$  is the number of tiles which have been eliminated in
       column  $j$  */
2    $nZ(j) = 0$ 
    /*  $nT(j)$  is the number of tiles which have been triangularized
       in column  $j$  */
3    $nT(j) = 0$ 
4 while column  $q$  is not finished do
5   for  $j = q$  down to 1 do
6     if  $j == 1$  then
7       /* Triangularize the first column if not yet done */
8        $nT_{\text{new}} = nT(j) + (p - nT(j))$ 
9       if  $p - nT(j) > 0$  then
10        for  $k = p$  down to 1 do
11           $GEQRT(k, j)$ 
12          for  $jj = j + 1$  to  $q$  do
13             $UNMQR(k, j, jj)$ 
14        else
15          /* Triangularize every tile having a zero in the
             previous column */
16           $nT_{\text{new}} = nZ(j - 1)$ 
17          for  $k = nT(j)$  to  $nT_{\text{new}} - 1$  do
18             $GEQRT(p - k, j)$ 
19            for  $jj = j + 1$  to  $q$  do
20               $UNMQR(p - k, j, jj)$ 
21          /* Eliminate every tile triangularized in the previous step
             */
22           $nZ_{\text{new}} = nZ(j) + \lfloor \frac{nT(j) - nZ(j)}{2} \rfloor$ 
23          for  $kk = nZ(j)$  to  $nZ_{\text{new}} - 1$  do
24             $piv(p - kk) = p - kk - nZ_{\text{new}} + nZ(j)$ 
25             $TTQRT(p - kk, piv(p - kk), j)$ 
26            for  $jj = j + 1$  to  $q$  do
27               $TTMQR(p - kk, piv(p - kk), j, jj)$ 
28          /* Update the number of triangularized and eliminated tiles
             at the next step */
29           $nT(j) = nT_{\text{new}}$ 
30           $nZ(j) = nZ_{\text{new}}$ 
```

Table 3.3: Time-steps for tiled algorithms.

(a) SAMEH-KUCK	(b) FIBONACCI	(c) GREEDY	(d) BINARYTREE	(e) PLASMA TREE ($BS = 5$)
★	★	★	★	★
6	14	12	6	6
8	12	10	8	8
28	48	42	28	28
★	★	★	★	★
10	12	10	6	10
34	46	40	36	34
50	70	64	56	50
★	★	★	★	★
12	10	8	10	12
40	42	36	34	40
56	68	62	70	56
72	92	86	90	72
★	★	★	★	★
14	10	8	6	14
46	40	34	44	46
62	64	56	68	62
78	90	84	104	78
94	114	106	124	94
★	★	★	★	★
16	10	8	8	16
52	40	34	28	52
68	62	56	78	68
84	86	78	102	84
100	112	102	138	100
116	136	122	158	116
★	★	★	★	★
18	8	6	6	18
58	36	30	42	58
74	62	52	62	74
90	84	78	112	90
106	108	100	136	106
122	134	122	172	122
★	★	★	★	★
20	8	6	12	20
64	34	28	40	64
80	58	50	76	80
96	84	72	96	96
112	106	100	146	112
128	130	118	170	128
★	★	★	★	★
22	8	6	6	22
70	34	28	46	70
86	56	50	74	86
102	80	72	110	102
118	106	94	130	118
134	128	116	180	134
★	★	★	★	★
24	8	6	8	24
76	34	28	28	76
92	56	50	80	92
108	78	68	108	108
124	102	94	144	124
140	128	116	164	140
★	★	★	★	★
26	6	6	6	26
82	28	24	36	82
98	56	44	56	98
114	78	66	114	114
130	100	88	142	130
146	122	110	178	146
★	★	★	★	★
28	6	6	10	28
88	28	22	34	88
104	56	44	64	104
120	78	60	84	120
136	100	82	148	136
152	122	104	176	152
★	★	★	★	★
30	6	6	6	30
94	28	22	38	94
110	56	44	62	110
126	78	60	92	126
142	100	82	112	142
158	122	104	182	158
★	★	★	★	★
32	6	6	8	32
100	22	22	28	100
116	44	38	66	116
132	60	60	90	132
148	94	76	114	148
164	116	98	134	164

Table 3.4: Neither GREEDY nor ASAP are optimal.

(a) GREEDY nor ASAP are optimal.

(a) GREEDY	(b) ASAP
★	★
12	12
★	★
10	10
42	40
★	★
10	10
40	36
64	86
8	8
36	34
62	80
8	8
34	32
56	74
8	8
34	30
56	68
8	8
30	28
52	62
6	6
28	28
50	56
6	6
28	26
50	50
6	6
28	24
50	46
6	6
28	24
44	44
6	6
22	22
44	44
6	6
22	22
44	40
6	6
22	22
38	38

(b) GREEDY generally outperforms ASAP.

p	Algorithm	q			
		16	32	64	128
16	GREEDY	310			
	ASAP	310			
32	GREEDY	360	650		
	ASAP	402	656		
64	GREEDY	374	726	1342	
	ASAP	588	844	1354	
128	GREEDY	396	748	1452	2732
	ASAP	966	1222	1748	2756

Table 3.5: Three schemes applied to a column whose update kernel weight is not an integer multiple of the elimination kernel weight.

a_{11}	(1)	(2)	(3)
6			
6	13	14	12
6	11	8	10
6	9	8	10
6	9	12	8
6	9	9	8
6	7	7	8
6	7	7	8
6	5	5	5
6	5	5	5
6	5	5	5

Table 3.6: Greedy versus PT-TT and Fibonacci (Theoretical)

p	q	GREEDY	PT-TT	BS	Overhead	Gain	Fib	Overhead	Gain
40	1	16	16	1	1.0000	0.0000	22	1.3750	0.2727
40	2	54	60	3	1.1111	0.1000	72	1.3333	0.2500
40	3	74	98	5	1.3243	0.2449	94	1.2703	0.2128
40	4	104	132	5	1.2692	0.2121	116	1.1154	0.1034
40	5	126	166	5	1.3175	0.2410	138	1.0952	0.0870
40	6	148	198	10	1.3378	0.2525	160	1.0811	0.0750
40	7	170	226	10	1.3294	0.2478	182	1.0706	0.0659
40	8	192	254	10	1.3229	0.2441	204	1.0625	0.0588
40	9	214	282	10	1.3178	0.2411	226	1.0561	0.0531
40	10	236	310	10	1.3136	0.2387	248	1.0508	0.0484
40	11	258	336	20	1.3023	0.2321	270	1.0465	0.0444
40	12	280	358	20	1.2786	0.2179	292	1.0429	0.0411
40	13	302	380	20	1.2583	0.2053	314	1.0397	0.0382
40	14	324	402	20	1.2407	0.1940	336	1.0370	0.0357
40	15	346	424	20	1.2254	0.1840	358	1.0347	0.0335
40	16	368	446	20	1.2120	0.1749	380	1.0326	0.0316
40	17	390	468	20	1.2000	0.1667	402	1.0308	0.0299
40	18	412	490	20	1.1893	0.1592	424	1.0291	0.0283
40	19	432	512	20	1.1852	0.1562	446	1.0324	0.0314
40	20	454	534	20	1.1762	0.1498	468	1.0308	0.0299
40	21	476	554	20	1.1639	0.1408	490	1.0294	0.0286
40	22	498	570	20	1.1446	0.1263	512	1.0281	0.0273
40	23	520	586	20	1.1269	0.1126	534	1.0269	0.0262
40	24	542	602	20	1.1107	0.0997	556	1.0258	0.0252
40	25	564	618	20	1.0957	0.0874	578	1.0248	0.0242
40	26	586	634	20	1.0819	0.0757	600	1.0239	0.0233
40	27	608	650	20	1.0691	0.0646	622	1.0230	0.0225
40	28	630	666	20	1.0571	0.0541	644	1.0222	0.0217
40	29	652	682	20	1.0460	0.0440	666	1.0215	0.0210
40	30	668	698	20	1.0449	0.0430	688	1.0299	0.0291
40	31	684	714	20	1.0439	0.0420	710	1.0380	0.0366
40	32	700	730	20	1.0429	0.0411	732	1.0457	0.0437
40	33	716	746	20	1.0419	0.0402	754	1.0531	0.0504
40	34	732	762	20	1.0410	0.0394	776	1.0601	0.0567
40	35	748	778	20	1.0401	0.0386	798	1.0668	0.0627
40	36	764	794	20	1.0393	0.0378	820	1.0733	0.0683
40	37	780	810	20	1.0385	0.0370	842	1.0795	0.0736
40	38	796	826	20	1.0377	0.0363	862	1.0829	0.0766
40	39	812	842	20	1.0369	0.0356	878	1.0813	0.0752
40	40	826	856	20	1.0363	0.0350	892	1.0799	0.0740

Table 3.7: Greedy versus PT-TT (Experimental Double)

p	q	GREEDY(d)	PT-TT(d)	BS	Overhead	Gain
40	1	36.9360	37.5020	1	1.0153	-0.0153
40	2	58.5090	52.7180	3	0.9010	0.0990
40	4	103.2670	90.7940	10	0.8792	0.1208
40	5	115.3060	100.5540	5	0.8721	0.1279
40	10	153.5180	145.8200	17	0.9499	0.0501
40	20	170.8730	171.8270	27	1.0056	-0.0056
40	40	184.5220	182.8160	19	0.9908	0.0092

Table 3.8: Greedy versus PT-TT (Experimental Double Complex)

p	q	GREEDY(z)	PT-TT(z)	BS	Overhead	Gain
40	1	42.0710	42.7120	1	1.0152	-0.0152
40	2	60.4420	52.1970	5	0.8636	0.1364
40	4	95.1820	84.1120	5	0.8837	0.1163
40	5	107.6370	96.7530	5	0.8989	0.1011
40	10	135.0270	128.4320	17	0.9512	0.0488
40	20	144.4010	146.4220	28	1.0140	-0.0140
40	40	152.9280	151.9090	8	0.9933	0.0067

Table 3.9: Greedy versus Fibonacci (Experimental Double)

p	q	GREEDY(d)	FIB(d)	Overhead	Gain
40	1	36.9360	26.5610	0.7191	0.2809
40	2	58.5090	49.4870	0.8458	0.1542
40	4	103.2670	100.1440	0.9698	0.0302
40	5	115.3060	115.0020	0.9974	0.0026
40	10	153.5180	152.0090	0.9902	0.0098
40	20	170.8730	170.4780	0.9977	0.0023
40	40	184.5220	180.2990	0.9771	0.0229

Table 3.10: Greedy versus Fibonacci (Experimental Double Complex)

p	q	GREEDY(z)	FIB(z)	Overhead	Gain
40	1	42.0710	30.2280	0.7185	0.2815
40	2	60.4420	48.9570	0.8100	0.1900
40	4	95.1820	97.1650	1.0208	-0.0208
40	5	107.6370	105.9610	0.9844	0.0156
40	10	135.0270	134.5500	0.9965	0.0035
40	20	144.4010	145.5530	1.0080	-0.0080
40	40	152.9280	150.0980	0.9815	0.0185

4. Scheduling of Cholesky Factorization

In Chapter 2 we studied the Cholesky Inversion algorithm which consists of the three steps: Cholesky factorization, inversion of the factor, and the multiplication of two triangular matrices. In this chapter, we will focus on the Cholesky factorization but unlike the previous work where the number of processors was unbounded, we will consider the factorization in the context of a finite number of processors. By limiting the number of processors, the scheduling of the tasks becomes an issue. Moreover, the weight (processing time) of each task must be taken into consideration when creating the schedule.

As before, we will be considering the critical path length for the algorithm but not as a function of the number of tiles rather as a function of the weights of the tasks. The weights are based upon the total computational cost for each kernel and are provided in Table 4.1. A more in-depth analysis of the length of the critical path with weighted tasks for the Cholesky Inversion algorithm can be found in [16] which also provides $9t - 10$ as the weighted critical path length for the Cholesky factorization of a matrix of $t \times t$ tiles.

Table 4.1: Task Weights

	# flops	Weights (in $\frac{1}{3}n_b^3$ flops)
POTRF	$\frac{1}{3}n_b^3 + O(n_b^2)$	1
TRSM	n_b^3	3
SYRK	$n_b^3 + O(n_b^2)$	3
GEMM	$2n_b^3 + O(n_b^2)$	6

The upper bound on performance of perfect speedup and critical path introduced in Chapter 2 remains too optimistic and does not take into account any information which can be garnered from the DAG of the algorithm. This work makes progress towards providing a more representative bound on the performance of the Cholesky factorization in the tiled setting.

We also provide gains toward a bound on the minimum number of processors required to obtain the shortest possible weighted critical path (minimum make span) for the Cholesky factorization for a matrix of $t \times t$ tiles.

4.1 ALAP Derived Performance Bound

To obtain our bounds, we calculate the latest possible start time for each task (ALAP) and consider an unbounded number of processors without any costs for communication. If we did account for communication, we might see the critical path length increase which would in turn decrease our upper bound. We start at the final tasks and consider how many processors are needed to execute these tasks without increasing the length of the critical path. We step backwards in time until such a point that there are more processors needed to keep the critical path length constant. Thus we must add enough processors to execute the tasks and in turn create more idle time for the execution of tasks which are successors. At a certain point, there is no more need to add processors and this is then the number of processors needed to obtain the constant length critical path.

By forcing as late as possible (ALAP) start times, any schedule will keep as many or fewer processors active as the ALAP execution on an unbounded number of processors. Thus by evaluating the Lost Area (LA), or idle time, for a given number of processors, p , at the end of the ALAP execution on an unbounded number of processors, we can increase the sequential time by the amount of LA and divide this result by the p to obtain the best possible execution time, i.e.,

$$T_p = \frac{T_{seq} + LA_p}{p} \quad (4.1)$$

and we define this to be the *ALAP Derived Performance Bound*. Hence the maximum speedup that we can expect is given by

$$T_{seq} \cdot \frac{p}{T_{seq} + LA_p}.$$

An example will help to further illustrate this technique. In Figure 4.1 we are given the ALAP execution of a 5×5 tiled matrix which has $T_{seq} = 125$. The ordered pairs indicated provide the number of processors and idle time, respectively, and in Table 4.2 are given the values for T_p , speedup, and efficiency. For more than four processors, there are enough processors to obtain the critical path length which becomes our limiting factor.

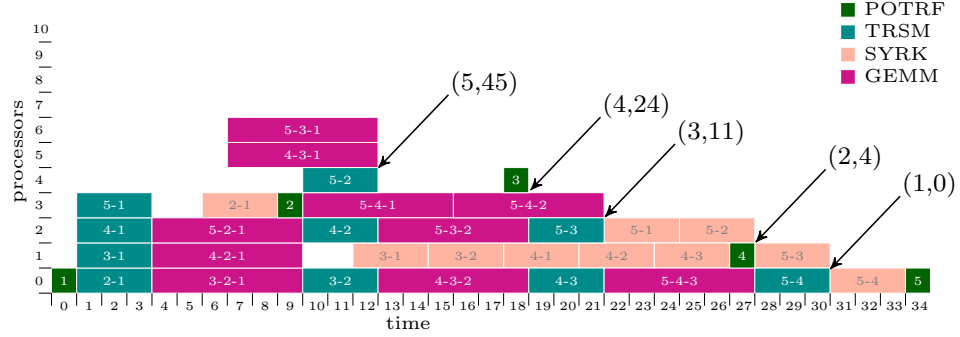


Figure 4.1: ALAP execution for 5×5 tiles

Table 4.2: Upper bound on speedup and efficiency for 5×5 tiles

p	T_p	S_p	E_p
1	125.00	1.00	1.00
2	64.50	1.94	0.97
3	45.33	2.76	0.92
4	37.25	3.36	0.84
5	35.00	3.57	0.71
6	35.00	3.57	0.60
7	35.00	3.57	0.51
8	35.00	3.57	0.45
9	35.00	3.57	0.40
10	35.00	3.57	0.36

4.2 Critical Path Scheduling

In order to provide a critical path schedule, we use the Backflow algorithm to assign priorities to tasks of a DAG such that each task's priority adheres to its de-

dependencies. The algorithm is described in four steps:

- STEP 1 : Beginning at the final task in the DAG, set its priority to its processing time.
- STEP 2 : Moving in the reverse direction, set each incidental task's priority to the sum of its the processing time and the final task's priority.
- STEP 3 : For each task in STEP 2, moving in the reverse direction, set each incidental task's priority to the sum of its processing time and the maximum priority of any incidental successor task.
- STEP 4 : Repeat the procedure until all tasks have been assigned a priority.

An example is given in Figure 4.2. The processing times are given in parenthesis and the assigned priorities (cp) are designated in square brackets. Tasks A and B will be assigned a priority of 16 since $cp(A) = 3 + \max(cp(C), cp(D))$ and $cp(B) = 3 + \max(cp(D), cp(E))$.

By following the path with the highest priorities, a critical path can be discerned from the weighted DAG. Thus any schedule which then chooses from the available tasks those with the highest priorities to execute first inherently follows the critical path. It is well known that a critical path scheduling is not always optimal. As an example [43], take two processors and four tasks. Let tasks A , B , C , and D have weights of 3, 3, 1, and 1, respectively and let the only relationship between tasks be that C is a predecessor of D . Then $cp(A) = cp(B) = 3$, $cp(C) = 2$, and $cp(D) = 1$. A critical path schedule would choose to schedule tasks A and B simultaneously and follow up with C and then D , resulting in a schedule of length five. However, if A and C are scheduled simultaneously and then D follows A on the same processor and B follows C on the other processor, the length of the schedule is four.

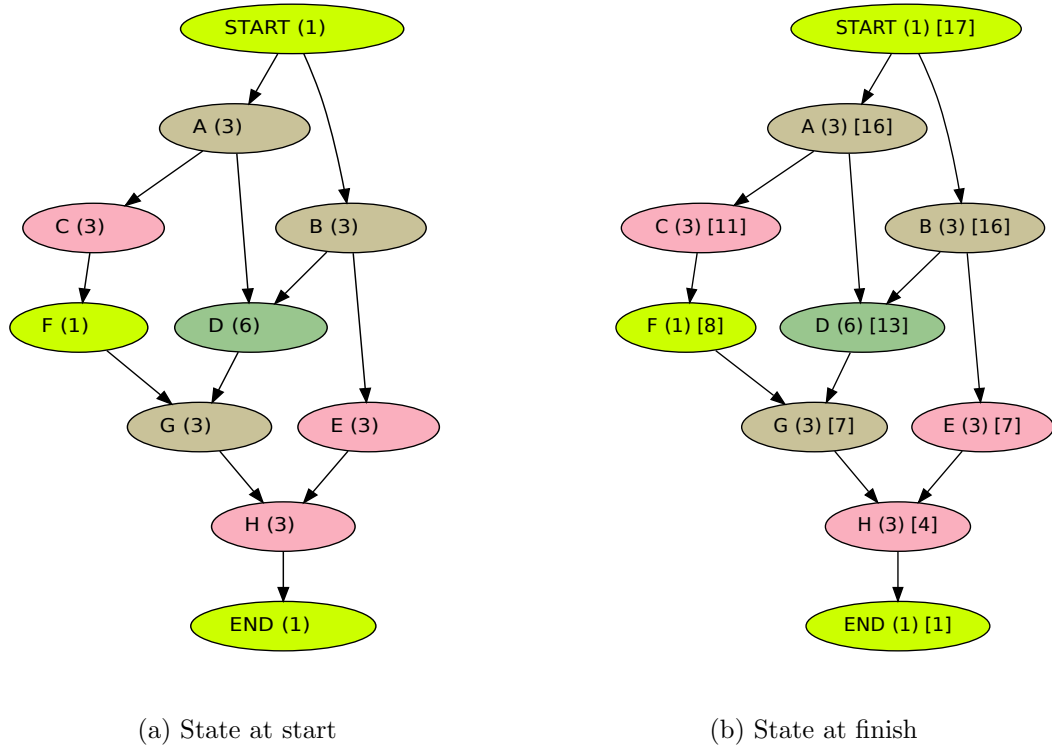


Figure 4.2: Example derivation of task priorities via the Backflow algorithm

We will use this critical path information to analyze three schedules by choosing available tasks via $\max(cp)$, $\text{rand}(cp)$, or $\min(cp)$. The $\max(cp)$ will naturally follow the critical path by scheduling tasks with the highest cp first and, vice versa, the $\min(cp)$ will schedule from the available tasks those with the minimum cp . Between these two, we also choose randomly amongst the available tasks with $\text{rand}(cp)$.

4.3 Scheduling with synchronizations

The right-looking version of the LAPACK Cholesky factorization as depicted in Figure 1.1c provides an alternative schedule which can be easier to analyze and understand. We will apply the three steps of the algorithm to a matrix of $t \times t$ tiles. In the tiled setting, we can provide synchronization points between the varying tasks of each step and simply schedule any of the available tasks since there are no dependencies between the tasks in each grouping. By adding these synchronizations,

this schedule is not able to obtain the critical path no matter how many processors are available. Algorithm 4.1 is the right-looking variant of the Cholesky factorization with added synchronization points.

Algorithm 4.1: Schedule for tiled right-looking Cholesky factorization with added synchronizations to allow for grouping.

```

1 Tile Cholesky Factorization (compute  $L$  such that  $A = LL^T$ );
2 for  $j = 0$  to  $t - 1$  do
3   schedule POTRF( $i$ ) ;
4   synchronize;
5   for  $j = i + 1$  to  $t - 1$  do
6     schedule TRSM( $j, i$ ) ;
7   synchronize;
8   for  $j = i + 1$  to  $t - 1$  do
9     for  $k = i + 1$  to  $j - 1$  do
10      schedule GEMM( $j, i, k$ ) ;
11   synchronize;
12   for  $j = i + 1$  to  $t - 1$  do
13     schedule SYRK( $j, i$ ) ;
14   synchronize;
```

Naturally, we can improve upon the above schedule by removing the synchronization between some of the groupings (Algorithm 4.2). The update of the trailing matrix is composed of two groupings, namely the GEMMs and the SYRKs, which can be executed in parallel if enough processors are available. Moreover, the added synchronization point between the update of the trailing matrix and the factorization of the next diagonal tile may also be removed. This schedule does become more complex, but given enough processors, the schedule is able to obtain the critical path as the limiting factor to performance. The minimum number of processors, p , needed to obtain the critical path is $p = \lceil \frac{1}{2}(t - 1)^2 \rceil$ for a matrix of $t \times t$ tiles since the highest degree of parallelism is realized for the update of the first trailing matrix which is of

size $(t - 1) \times (t - 1)$.

Both of these schedules differ from the critical path scheduling due to the added synchronization points and will show lower theoretical performance. In the theoretical results, we only show Algorithm 4.2.

Algorithm 4.2: Improvement upon Algorithm 4.1

```

1 Tile Cholesky Factorization (compute L such that A = LLT);
2 for  $j = 0$  to  $t - 1$  do
3   schedule POTRF(i) ;
4   synchronize;
5   for  $j = i + 1$  to  $t - 1$  do
6     schedule TRSM(j,i) ;
7   synchronize;
8   for  $j = i + 1$  to  $t - 1$  do
9     for  $k = i + 1$  to  $j - 1$  do
10      schedule GEMM(j,i,k) ;
11      schedule SYRK(j,i) ;
```

4.4 Theoretical Results

In the following figures, our *Rooftop* bound will be that which uses the perfect speedup until the weighted critical path is the limiting factor, i.e.,

$$Rooftop\ Bound = \max \left(\text{Critical Path}, \frac{T_{seq}}{p} \right) \quad (4.2)$$

We will compare this to our *ALAP Derived* bound which was derived using the ALAP execution, our various scheduling strategies, and the following lower bound. From [20, p.221,§7.4.2], given our DAG, we know that the make span, MS , of any list schedule, σ , for a given number of processors, p , is

$$MS(\sigma, p) \leq \left(2 - \frac{1}{p} \right) MS_{opt}(p)$$

where MS_{opt} is the make span of the optimal list schedule without communication costs. However, we do not know MS_{opt} and must therefore substitute the make span

of the Critical Path Scheduling using the maximum strategy to compute our lower bound.

The ALAP Derived bound improves upon the Rooftop bound precisely in the area that is of most concern, namely where there is enough parallelism but not enough processors to fully exploit that parallelism.

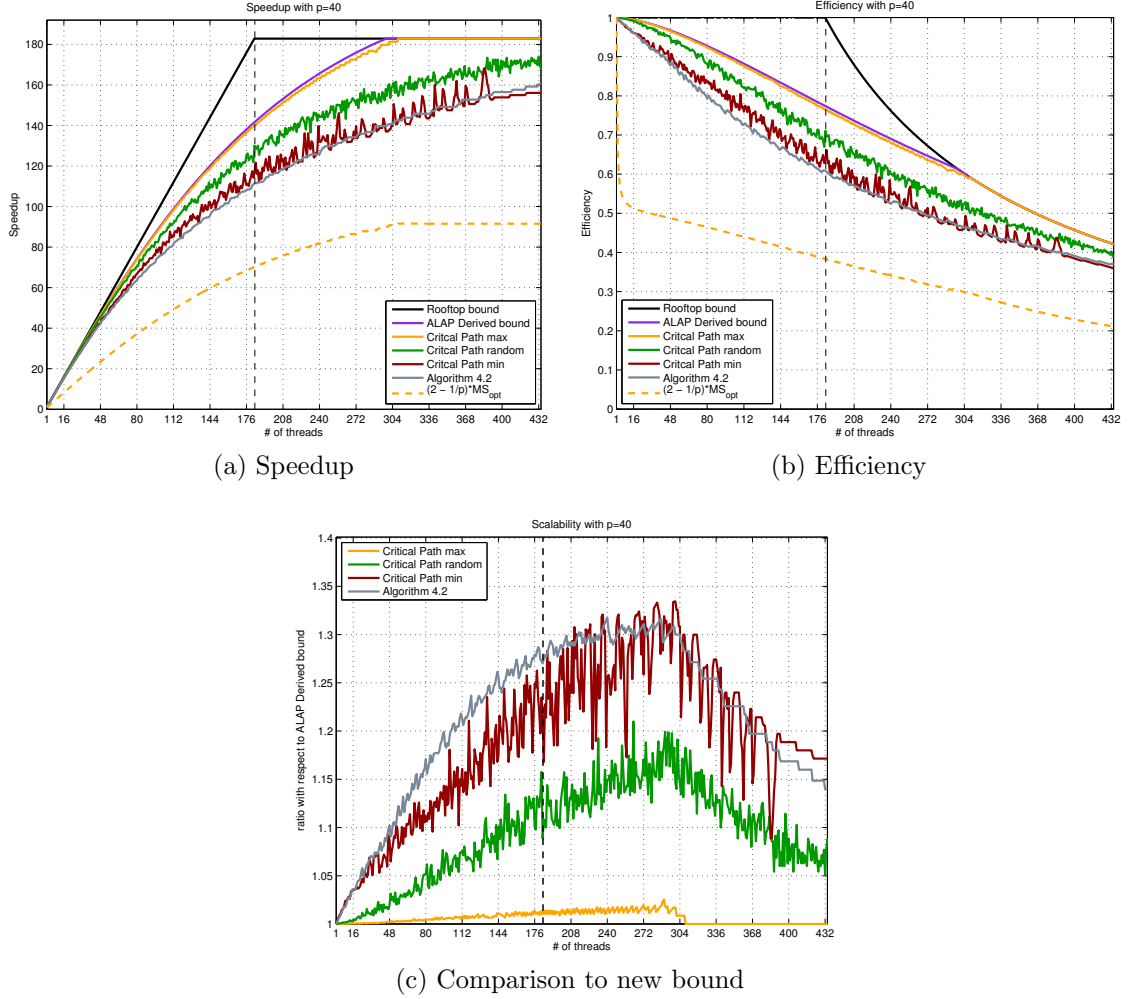


Figure 4.3: Theoretical results for matrix of 40×40 tiles.

Figure 4.3(a) shows that the Critical Path schedule is actually quite decent and comes to within two percent of the ALAP Derived bound. Moreover, the ALAP Derived bound has significantly reduced the gap between the Rooftop bound and any of our list schedules.

4.5 Toward an α_{opt}

It is interesting to know how many processors one needs to be able to schedule all of the tasks and maintain the weighted critical path. We will view this problem in terms of tiles and state the problem as follows:

Given a matrix of $t \times t$ tiles, determine the minimum number of processors, p_{opt} , needed to schedule all tasks and achieve an execution time equal to the weighted critical path.

Toward that end, we will let $p = \alpha t^2$ where $0 < \alpha \leq 1$. Our analysis will be asymptotic in which we let t tend to infinity. From the analysis of Algorithm 4.2, we already know that $\alpha_{opt} \leq \frac{1}{2}$. Using MATLAB, we have calculated the α value for matrices of $t \times t$ tiles as shown in Figure 4.4. It is our conjecture that $\alpha_{opt} \approx \frac{1}{5}$.

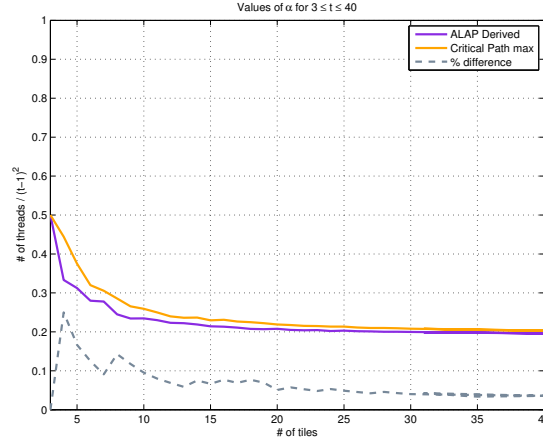


Figure 4.4: Values of α for matrices of $t \times t$ tiles where $3 \leq t \leq 40$

4.6 Related Work

For the LU decomposition with partial pivoting, much work has been accomplished to discern asymptotically optimal algorithms for all values of α [37, 35, 43]. They consider a problem of size n and assume $p = \alpha n$ processors on which to schedule the LU decomposition. The critical path length of the optimal schedule in this case

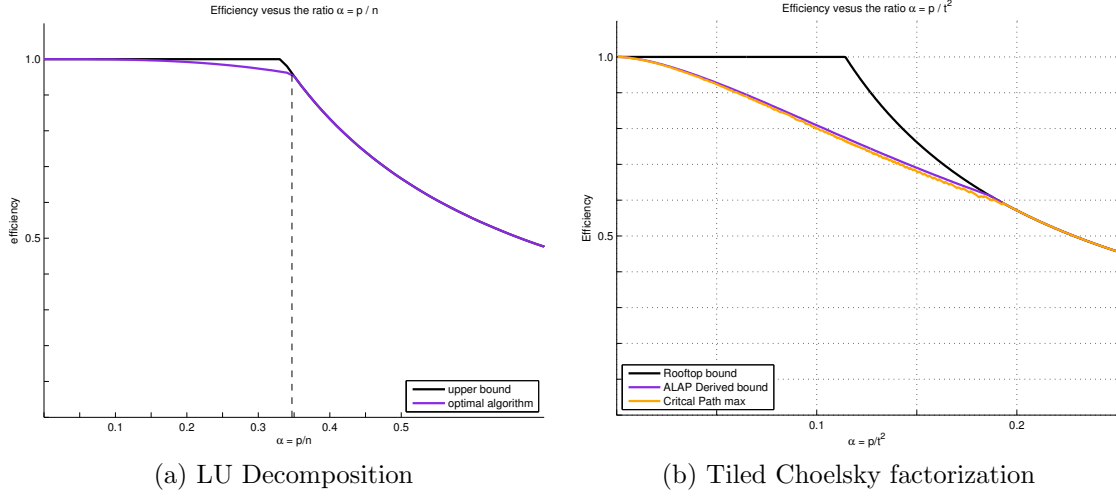


Figure 4.5: Asymptotic efficiency versus $\alpha = p/n$ for LU decomposition and versus $\alpha = p/t^2$ for Tiled Cholesky factorization.

is n^2 and $\alpha_{opt} \approx 0.347$ where α_{opt} is a solution to the equation $3\alpha - \alpha^3 = 1$.

In Figure 4.5, we make a comparison between the algorithmic efficiency of the LU decomposition and the tiled Cholesky factorization. In the case of the LU decomposition, the attainable upper bound on efficiency closely resembles our previous bound of perfect speedup which is then limited by the critical path. On the other hand, the tiled Cholesky factorization does not exhibit this type of efficiency which can be seen from the gap between our Rooftop bound and the ALAP Derived bound. Unlike the work in [37], we do not provide an algorithm which attains the ALAP Derived bound.

4.7 Conclusion and future work

In many research papers, the performance of an algorithm is usually compared to either the performance of the GEMM kernel or against perfect scalability resulting in large discrepancies between the peak performance of the algorithm and these upper bounds. If an algorithm displays a DAG such as that of the tiled Cholesky factorization, it is unrealistic to expect perfect scalability or even make comparisons to the performance of the GEMM kernel. Thus one needs to consider a new bound which is more representative of the algorithm and accounts for the structure of the DAG. Without such a bound it is difficult to assess whether there are any performance

gains to be achieved. Although we do not have a closed-form expression for this new bound, we have shown that such a bound exists. Moreover, we have also shown that any algorithm which schedules the tiled Cholesky factorization while maintaining the weighted critical path will require $O(t^2)$ processors for a matrix of $t \times t$ tiles and the coefficient is somewhere around 0.2.

In this chapter, we have applied a combination of existing techniques to a tiled Cholesky factorization algorithm to discover a more realistic bound on the performance and efficiency. We did so by considering an ALAP execution on an unbounded number of processors and used this information to calculate the idle time for any list schedule on a bounded number of processors. This is then used to calculate the maximum speedup and efficiency that may be observed.

Further work is necessary to provide a closed-form expression of the new bound dependent upon the number of processors. In addition, we need to include communication costs in the bound to make it more reflective of the actual scheduling problem on parallel machines. As can be seen in Figure 4.3(c), the Critical Path Schedule is within 2% of our ALAP Derived bound. Although scheduling a DAG on a bounded number of processors is an NP-complete problem, it may be not be the case for the DAG of the tiled Cholesky factorization. Pursuant investigation might show that the Critical Path Scheduling is the optimal schedule.

5. Scheduling of QR Factorization

In this chapter, we present collaborative work with Jeffrey Larson. We revisit the tiled QR factorization as presented in Chapter 3 but do so in the context of a bounded number of resources. (Chapter 3 was concerned with finding the optimal elimination tree on an unlimited number of processors.) We will be using the same analytical tools as in Chapter 4 to derive good schedules and to improve upon the Rooftop bound. The Cholesky factorization has just one DAG, therefore Chapter 4 is a standard scheduling problem, i.e., how to schedule a DAG on a finite number of processor. Unlike the previous chapter, we will need to consider all of the various algorithms (i.e., elimination trees) and cannot distill the analysis down to a single DAG.

5.1 Performance Bounds

Each of the algorithms studied in Chapter 3, namely FLATTREE, FIBONACCI, GREEDY, and GRASAP, produces a unique DAG for a matrix of $p \times q$ tiles. In turn, the ALAP Derived bounds (4.1) for each elimination tree will also be unique. In Figure 5.1, we give the computed upper bounds and make comparisons to the scheduling strategies of maximum, random, and minimum via the Critical Path Method for a matrix of 20×6 tiles. The matrix size was chosen such that the critical path length of GREEDY is 136 and the critical path length of GRASAP is 134 (see Figure 3.5 in § 3.2.2).

The GRASAP algorithm for a tiled matrix is optimal in the scope of unbounded resources. However by the manner in which the ALAP Derived bound is computed, the bound created by using GRASAP cannot hold for all of the other algorithms. Consider the ALAP execution of the FIBONACCI and GRASAP algorithms on a matrix of 15×4 tiles. In Figure 5.2, we show the execution of the last tasks for GRASAP on the left and FIBONACCI on the right. More of the tasks in the ALAP execution for FIBONACCI are pushed towards the end of the execution which means

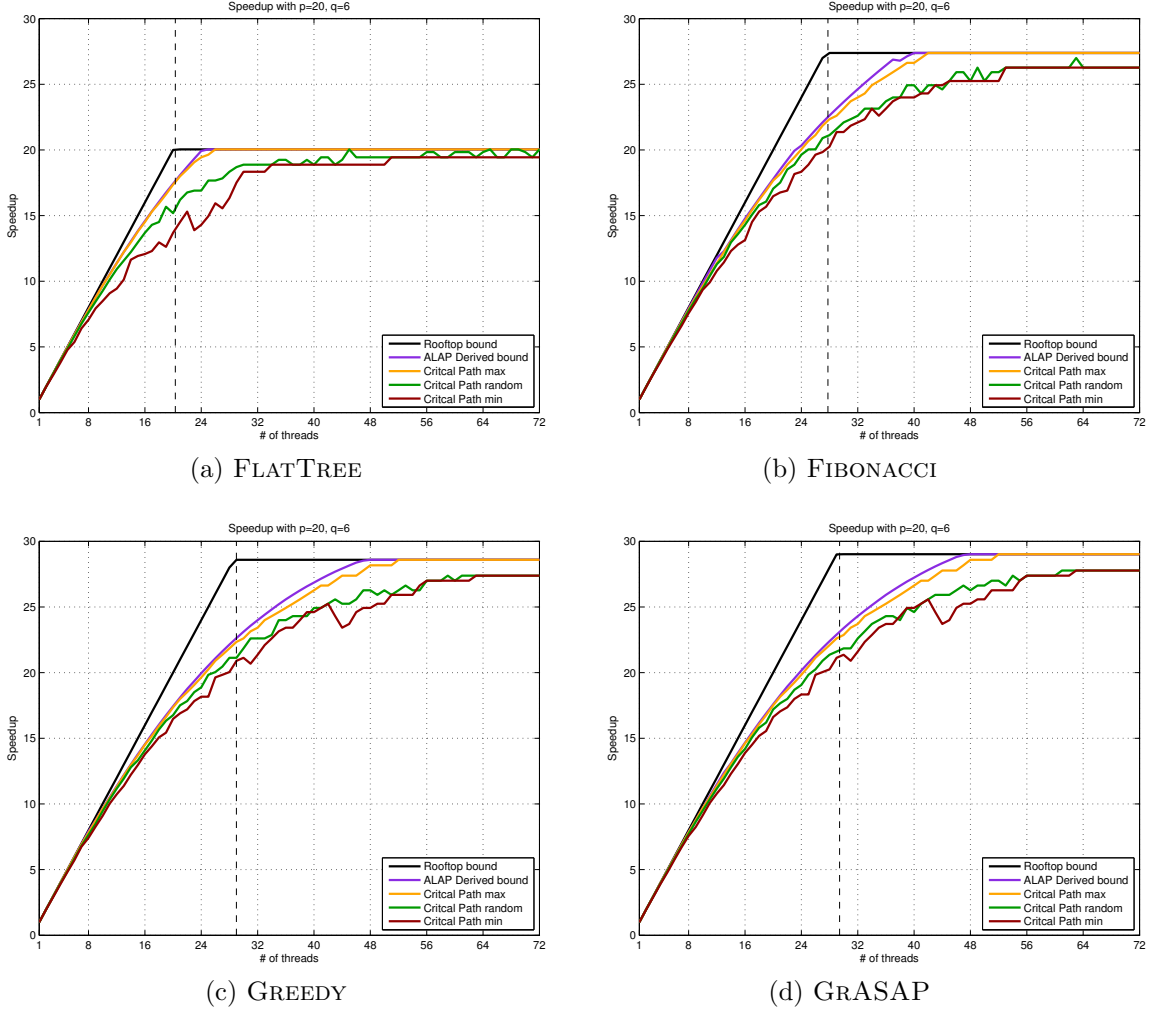


Figure 5.1: Scheduling comparisons for each of the algorithms versus the ALAP Derived bounds on a matrix of 20×6 tiles.

the ALAP Derived bound will be higher than that of GRASAP for a schedule that uses fewer than 10 processors. In other words, as we add more processors, the Lost Area (LA) increases much faster for GRASAP than it does for FIBONACCI. Since the critical path length for FIBONACCI is greater than that of GRASAP, after a certain number of processors, the ALAP Derived bound for FIBONACCI falls below that of GRASAP. These observations are evident in Figure 5.3 where we show a comparison of the bound for each algorithm. Recall that the Rooftop bound (4.2) only takes into account the critical path length of an algorithm such that for GRASAP

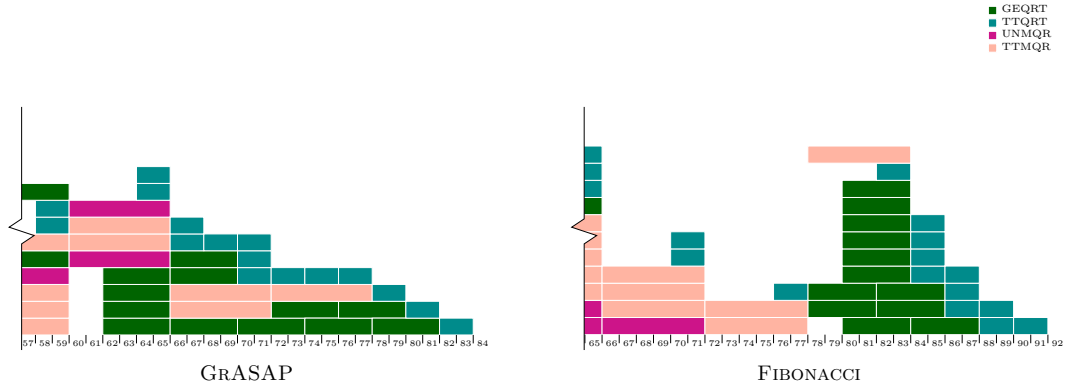


Figure 5.2: Tail-end execution using ALAP on unbounded resources for GRASAP and FIBONACCI on a matrix of 15×4 tiles.

it can be considered a bound for all the algorithms since GRASAP is optimal for unlimited resources and thus has the shortest critical path length.

5.2 Optimality

There is no reason for the tree found optimal in Chapter 3 on an unbounded number of resources to be optimal on a bounded number resources. We cast the problem of finding the optimal tree with the optimal schedule as an integer programming problem. (A complete description of the formulation can be found in Appendix A.) Similarly, in [1] a Mixed-Integer Linear Programming approach was used to provide an upper bound on performance. However, the integer programming problem size grows exponentially as the matrix size increases, thus the largest feasible problem size was a matrix of 5×5 tiles. In Figure 5.4 we show the speedup of the GRASAP algorithm with its bound and make comparisons to an optimal tree with an optimal schedule and Table 5.1 provides the actual schedule lengths for all of the algorithms using the CP Method for the matrix of 5×5 tiles..

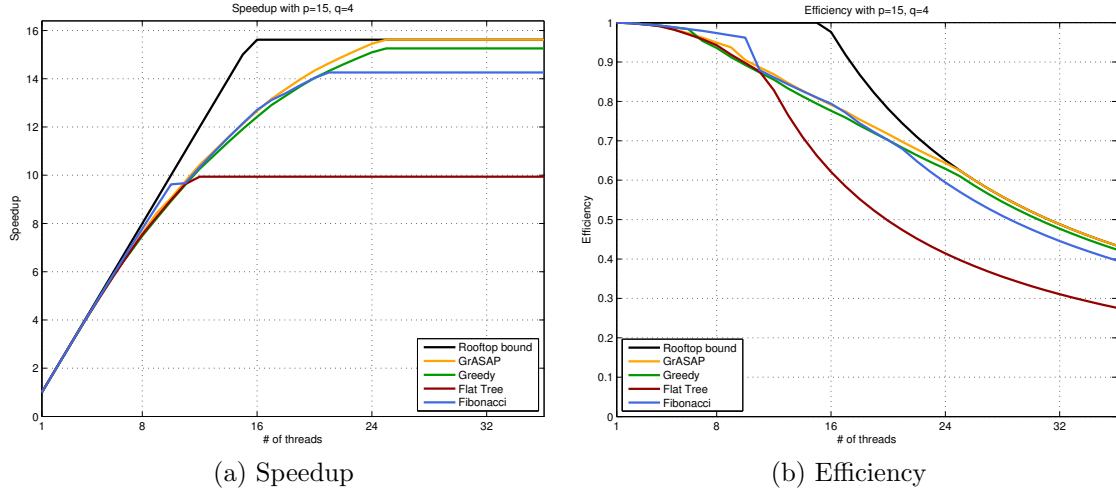


Figure 5.3: ALAP Derived bound comparison for all algorithms for a matrix of 15×4 tiles.

5.3 Elimination Tree Scheduling

We pair up the choice of the elimination tree with a type of scheduling strategy to obtain the following bounds:

$$\left(\begin{array}{c} \text{GRASAP} \\ \text{Rooftop bound} \end{array} \right) \leq \left(\begin{array}{c} \text{optimal tree} \\ \text{optimal schedule} \end{array} \right) \leq \left(\begin{array}{c} \text{GRASAP} \\ \text{optimal schedule} \end{array} \right) \leq \left(\begin{array}{c} \text{GRASAP} \\ \text{CP schedule} \end{array} \right)$$

Moreover, we also have

$$\left(\begin{array}{c} \text{GRASAP} \\ \text{Rooftop bound} \end{array} \right) \leq \left(\begin{array}{c} \text{GRASAP} \\ \text{ALAP Derived Bound} \end{array} \right) \leq \left(\begin{array}{c} \text{GRASAP} \\ \text{optimal schedule} \end{array} \right) \leq \left(\begin{array}{c} \text{GRASAP} \\ \text{CP schedule} \end{array} \right)$$

Combining these inequalities with Table 5.1 gives rise to the following questions:

Given an optimal elimination tree for the tiled QR factorization on an unbounded number of resources

(Q1) *does there always exist a scheduling strategy such that the schedule on limited resources is optimal?*

(Q2) *does the ALAP Derived bound for this elimination tree hold true for any scheduling strategy on any other elimination tree?*

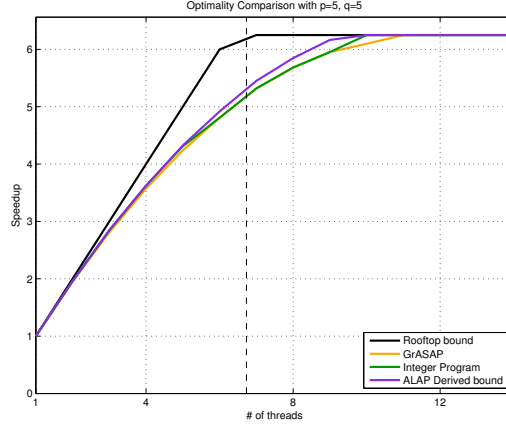


Figure 5.4: Comparison of speedup for CP Method on GRASAP, ALAP Derived bound from GRASAP, and optimal schedules for a matrix of 5×5 tiles on 1 to 14 processors

Table 5.1: Schedule lengths for matrix of 5×5 tiles

Procs	ALAP Derived	Optimal	CP Method			
	Bound(GRASAP)	Tree/Schedule	GRASAP	GREEDY	FIBONACCI	FLATTREE
1	500	500	500	500	500	500
2	255	256	256	256	256	256
3	176	176	178	178	178	176
4	138	138	140	140	140	140
5	116	116	118	118	118	116
6	102	104	104	104	104	104
7	92	94	94	94	94	94
8	86	88	88	88	88	88
9	82	84	84	84	86	86
10	80	80	82	82	86	86
11	80	80	80	80	86	86
12	80	80	80	80	86	86
13	80	80	80	80	86	86
14	80	80	80	80	80	86

From Chapter 3 we have that GRASAP is an optimal elimination tree for the tiled QR factorization. We know that the length of an optimal schedule for GRASAP on p processors will necessarily be greater or equal to the ALAP Derived bound for GRASAP on p processors by way of construction of the ALAP Derived bound. Thus (Q1) implies (Q2). We cannot address the first question directly since the size of the matrix needed to produce a counter example is too large for verification with the integer programming formulation.

Therefore we need to find a matrix size for which a schedule exists whose execution length is smaller than the ALAP Derived bound from GRASAP on the same matrix. As we have seen in Figure 5.3, the FIBONACCI elimination tree on a tall and skinny tiled matrix provides the best hope.

Consider a matrix of 34×4 tiles on 10 processors. The ALAP Derived bound from GRASAP is 188. Using the Critical Path method to schedule the FIBONACCI elimination tree we obtain a schedule length of 184. Therefore the ALAP Derived bound from GRASAP does not hold for this schedule. By implication, we have that (Q1) is false. However, the Rooftop bound from GRASAP is still a valid bound for all of the schedules.

5.4 Conclusion

In this chapter we have applied the same tools used in Chapter 4 to provide performance bounds for the tiled QR factorization. Further, we have shown that the ALAP Derived bound is algorithm dependent. This leaves that only bound we have for all algorithms is the Rooftop bound as computed using the GRASAP algorithm.

The analysis in this chapter has also shown that an optimal algorithm for an unbounded number of resources does not imply that a scheduling strategy exists such that it can be scheduled optimally.

6. Strassen Matrix-Matrix Multiplication

Matrix multiplication is the underlying operation in many if not most of the applications in numerical linear algebra and as such, it has garnered much attention. Algorithms such as the Cholesky factorization, LU decomposition and more recently the QR-based Dynamically Weighted Halley iteration for polar decomposition [38], spend a majority of their computational cost in matrix-matrix multiplication. The conventional BLAS Level 3 subprogram for matrix-matrix multiplication is of $O(n^\alpha)$, where $\alpha = \log_2 8 = 3$, computational cost but there exist subcubic computational cost algorithms. In 1969, Volker Strassen [48] presented an algorithm that computes the product of two square matrices of size $n \times n$, where n is even, using only 7 matrix multiplications at the cost of needing 18 matrix additions/subtractions which then can be called recursively for each of the 7 multiplications. This compares to the standard cubic algorithm which requires 8 matrix multiplications and only 4 matrix additions. When Strassen's algorithm is applied recursively down to a constant size, the computational cost is $O(n^\alpha)$ where $\alpha = \log_2 7 \approx 2.807$. Two years later, Shmuel Winograd proved that a minimal of 7 matrix multiplications and 15 matrix additions/subtractions, which is less than the 18 of Strassen's, are required for the product of two 2×2 matrices, see [54]. These discoveries have spawned a flurry of research over the years. In 1978, Pan [39] showed that $\alpha < 2.796$. In the late 1970's and early 1980's, Bini [12] provided $\alpha < 2.78$ with Schönage [46] following up by showing $\alpha < 2.522$ but was usurped the following year by Romani [44] who discerned $\alpha < 2.517$. In 1986, Strassen brought forth a new approach which lead to $\alpha < 2.497$. In 1990, Coppersmith and Winograd [23] improved upon Strassen's result providing the asymptotic exponent $\alpha < 2.376$. This final result still stands, but it is conjectured that $\alpha = 2 + \varepsilon$ for any $\varepsilon > 0$ where ε can be made as small as possible. Although the Coppersmith-Winograd algorithm may be reasonable to implement, since the constant of the algorithm is huge and will not provide an advantage except for very

large matrices, we will not consider it and instead focus on the Strassen-Winograd Algorithm.

6.1 Strassen-Winograd Algorithm

Here we discuss the algorithm as it would be implemented to compute the product of two matrices A and B where the result is stored into matrix C . The algorithm is recursive, thus we describe one step. Given the input matrices A , B , and C , divide them into four submatrices,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

then the 7 matrix multiplications and 15 matrix additions/subtractions are computed as depicted in Table 6.1 and Figure 6.1 shows the task graph of the Strassen-Winograd algorithm for one level of recursion.

Table 6.1: Strassen-Winograd Algorithm

Phase 1	$T_1 = A_{21} + A_{22}$		$T_5 = B_{12} - B_{11}$	
	$T_2 = T_1 - A_{11}$		$T_6 = B_{22} - T_5$	
	$T_3 = A_{11} - A_{21}$		$T_7 = B_{22} - B_{12}$	
	$T_4 = A_{12} - T_2$		$T_8 = T_6 - B_{21}$	
Phase 2	$Q_1 = T_2 \times T_6$		$Q_5 = T_1 \times T_5$	
	$Q_2 = A_{11} \times B_{11}$		$Q_6 = T_4 \times B_{22}$	
	$Q_3 = A_{12} \times B_{21}$		$Q_7 = A_{22} \times T_8$	
	$Q_4 = T_3 \times T_7$			
Phase 3	$U_1 = Q_1 + Q_2$		$C_{11} = Q_2 + Q_3$	
	$U_2 = U_1 + Q_4$		$C_{12} = U_1 + U_3$	
	$U_3 = Q_5 + Q_6$		$C_{21} = U_2 - Q_7$	
			$C_{22} = U_2 + Q_5$	

In essence, Strassen's approach is very similar to the observation that Gauss made concerning the multiplication of two complex numbers. The product of $(a + bi)(c + di) = ac - bd + (bc + ad)i$ would naively take four multiplications, but can actually be

Phase 3 of the Strassen-Winograd algorithm, a similar approach is used which is also executed in parallel.

Algorithm 6.1: Tiled Matrix Multiplication (tiled_gemm)

```

/* Input:   $n \times n$  tiled matrices  $A$  and  $B$ , Output:   $n \times n$  tiled
   matrix  $C$  such that  $C = A \times B$  */
1 for i = 1 to n do
2   for j = 1 to n do
3     for k = 1 to n do
4        $C_{ij} \leftarrow A_{ik} \times B_{kj} + C_{ij}$ 

```

If the cutoff for the recursion occurs before the tile level, the computation for each C_{ij} can be executed in parallel. Therefore our tiled implementation of the Strassen-Winograd algorithm exploits two levels of parallelism. Moreover, this allows some parts of the matrix multiplications to occur early on as can be seen in Figure 6.2 which shows the DAG for a matrix of 4×4 tiles with one level of recursion. Both Figure 6.2 and Figure 6.1 illustrate one level of recursion but the tiled task graph of a 4×4 tiled matrix clearly portrays the high degree of parallelism.

The conventional matrix-matrix multiplication algorithm requires 8 multiplications and 4 additions whereas the Strassen-Winograd algorithm requires 7 multiplications and 15 additions/subtractions for each level of recursion. Therefore, there are more tasks for the Strassen-Winograd algorithm as compared to the conventional matrix-matrix multiplication and it would behoove us to reduce the number of tasks which would also reduce the algorithmic complexity. On the other hand, since we are reducing the number of multiplications, the computational cost is also reduced since this requires a cubic operation versus the quadratic operation of the addition/subtractions.

The total number of tasks, T , of the Strassen-Winograd algorithm is given by

$$T = 7^r \left(\frac{p}{2^r}\right)^3 + 15 \sum_{i=0}^{r-1} 7^{r-i-1} \left(\frac{p}{2^{r-i}}\right)^2 = p^3 \left(\frac{7}{8}\right)^r + 5p^2 \left(\left(\frac{7}{4}\right)^r - 1\right)$$

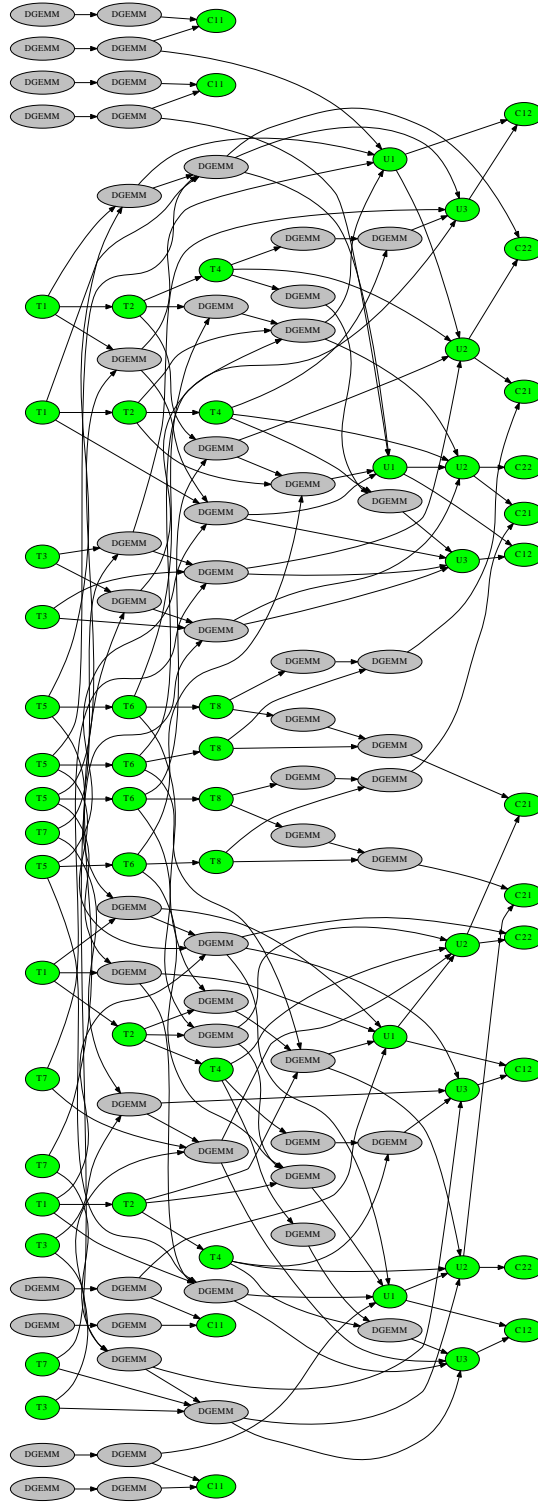


Figure 6.2: Strassen-Winograd DAG for matrix of 4×4 tiles with one recursion. Execution time progresses from left to right. Large ovals depict multiplication and small ovals addition/subtraction.

Table 6.2: Recursion levels which minimize the number of tasks for a tiled matrix of size $p \times p$

p	r_{min}	Gflop(SW)	Gflop(GEMM)
4	1	8.96e-01	1.02e+00
8	1	7.15e+00	8.18e+00
16	1	5.72e+01	6.55e+01
32	1	4.57e+02	5.24e+02
64	2	3.20e+03	4.19e+03
128	3	2.24e+04	3.35e+04
256	4	1.57e+05	2.68e+05
512	5	1.09e+06	2.14e+06
1024	6	7.69e+06	1.71e+07

which is minimized at recursion level r_{min} when

$$r_{min} = \left\lceil \left(\frac{\ln \left(\frac{p \ln \left(\frac{8}{7} \right)}{5 \ln \left(\frac{7}{4} \right)} \right)}{\ln 2} \right) \right\rceil,$$

and the total number of flops, F , is given by

$$F = m7^r \left(\frac{p}{2^r} \right)^3 + 15a \sum_{i=0}^{r-1} 7^{r-i-1} \left(\frac{p}{2^{r-i}} \right)^2 = mp^3 \left(\frac{7}{8} \right)^r + 5ap^2 \left(\left(\frac{7}{4} \right)^r - 1 \right),$$

where $m = 2n_b^3 - n_b^2$ for the multiplications and $a = n_b^2$ for the additions/subtractions.

As we increase the recursion, the number of tasks will decrease up to a certain point, r_{min} . The reason for this being at each recursion we reduce the number of p^3 tasks by $\frac{1}{8}$ while increasing the p^2 tasks by 15.

In our experiments, letting the tile size $n_b = 200$ provided the best performance. Thus Table 6.2 shows the corresponding values for the minimizing recursion level for various number of tiles. However, as can be seen in Table 6.3, the r_{min} which provides the minimum number of tasks does not provide the least amount of computational cost. The computational costs will be minimized at the full recursion.

Even though the number of tasks and thereby the computational complexity are minimized for r_{min} , Table 6.4 shows that the critical path length increases exponen-

Table 6.3: 128×128 tiles of size $n_b = 200$

algorithm	recursion	tasks	Gflop
strassen_winograd	1	1,896,448	2.92e+04
	2	1,774,592	2.56e+04
	3	1,762,048	2.24e+04
	4	1,915,712	1.96e+04
	5	2,338,288	1.72e+04
	6	3,212,252	1.51e+04
	7	4,859,338	1.33e+04
tilted_DGEMM		4,177,920	3.36e+04

Table 6.4: Comparison of the total number of tasks and critical path length for matrix of $p \times p$ tiles.

p	r	# tasks	CP	Ratio
4	0	64	2	32.0
	1	116	7	16.6
8	0	512	3	170.7
	1	688	9	76.4
	2	1,052	52	20.2
16	0	4,096	4	1,023.5
	1	4,544	13	349.5
	2	5,776	66	87.5
	3	8,324	361	23.1
32	0	32,768	5	6,553.6
	1	32,512	21	1,548.2
	2	35,648	94	379.2
	3	44,272	459	96.5
	4	62,108	2,524	24.6
64	0	262,144	6	43,690.7
	1	244,736	37	6,614.5
	2	242,944	150	1,619.6
	3	264,896	655	404.4
	4	325,264	3,210	101.3

tially with the recursion levels. Thus the amount of parallelism is likewise reduced for each recursion level.

6.3 Related Work

In practice the Strassen-Winograd algorithm imparts a large amount of overhead for small matrices, thus it is customary to overcome this issue by using it in conjunction with a conventional matrix multiplication operation. The key idea is to provide a recursion cutoff point such that once this is reached, the algorithm switches from calling itself recursively to calling, e.g., the BLAS Level 3 matrix multiplication. The recursion cutoff point is a tuning parameter which depends upon the machine architecture and can be either dynamic or static. In [49], one level of recursion is used while Chou [21] provides two levels of recursion by explicitly coding the 49 matrix multiplications which then is processed by either 7 or 49 processors. Our approach is to provide the recursion cutoff point as a parameter which can be set by the user.

There are various methods which can be used to deal with non square matrices and/or matrices of odd order. Methods such as static padding, dynamic padding, and dynamic peeling all provide these mechanisms. In this chapter, we only study matrices of tile order $p = 2^k$, i.e., $n = 2^k \cdot n_b$.

In [17], Boyer et al. propose schedules for both in-place and out-of-place implementations without the need for extra computations. Discovery of these algorithms was accomplished by using an exhaustive search algorithm. Their out-of-place algorithm makes use of the resultant matrix for temporary storage for the intermediate computations. This introduces unwanted data dependencies and more data movement leading to a loss of parallelism. Hence, we do not consider their out-of-place algorithm and focus on the classical Strassen-Winograd algorithm by using temporary storage allocations for all of the intermediate computations. As an example, given a tiled matrix of 128×128 tiles of size 200×200 , the input matrices and resultant matrix require 1.96608 GB of storage and allowing full recursion for our algorithm

Algorithm 6.2: Tiled Strassen-Winograd (tiled_gesw)

```
/* p is equal to the recursion cutoff, do the multiplication */
1 if p = r then
2   tiled_gemm( p, A, B, C )
3 else
4   /* p is greater than the recursion cutoff, so we split the
   problem in half */
5   p = p/2
6   /* Phase 1 */
7   tiled_geadd( p, A21, A22, T1 )
8   tiled_geadd( p, T1, A11, T2 )
9   tiled_geadd( p, A11, A21, T3 )
10  tiled_geadd( p, A12, T2, T4 )
11  tiled_geadd( p, B12, B11, T5 )
12  tiled_geadd( p, B22, T5, T6 )
13  tiled_geadd( p, B22, B12, T7 )
14  tiled_geadd( p, T6, B21, T8 )
15  /* Phase 2 */
16  tiled_gesw( p, T2, T6, Q1 )
17  tiled_gesw( p, A11, B11, Q2 )
18  tiled_gesw( p, A12, B21, Q3 )
19  tiled_gesw( p, T3, T7, Q4 )
20  tiled_gesw( p, T1, T5, Q5 )
21  tiled_gesw( p, T4, B22, Q6 )
22  tiled_gesw( p, A22, T8, Q7 )
23  /* Phase 3 */
24  tiled_geadd( p, Q1, Q2, U1 )
25  tiled_geadd( p, U1, Q4, U1 )
26  tiled_geadd( p, Q5, Q6, U1 )
27  tiled_geadd( p, Q2, Q3, C11 )
28  tiled_geadd( p, U1, U3, C12 )
29  tiled_geadd( p, U2, Q7, C21 )
30  tiled_geadd( p, U2, Q5, C22 )
```

would require an additional 3.2766 GB of temporary storage (see Figure 6.3).

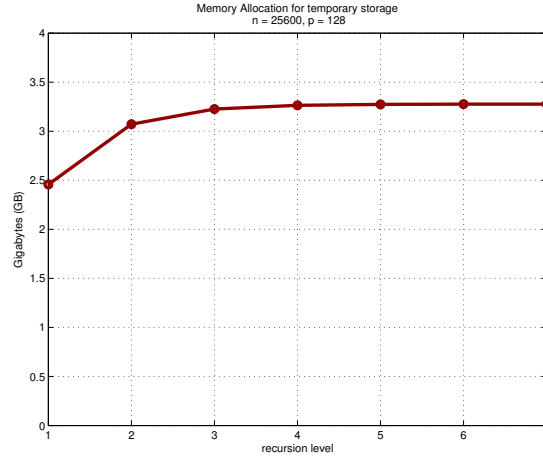


Figure 6.3: Required extra memory allocation for temporary storage for varying recursion levels.

In [27] a sequential implementation of Strassen-Winograd is studied by Douglas et al. which also provides means for a hybrid parallel implementation where the lower level is the sequential Strassen-Winograd and the upper level algorithm is the standard subdivided matrix multiplication. Comparisons are made to sequential implementations available in the IBM’s ESSL and Cray’s Scientific and Math library. Although this would have been an interesting project in and of itself within a tiled framework, the emphasis in this chapter was to provide the Strassen-Winograd at the upper level.

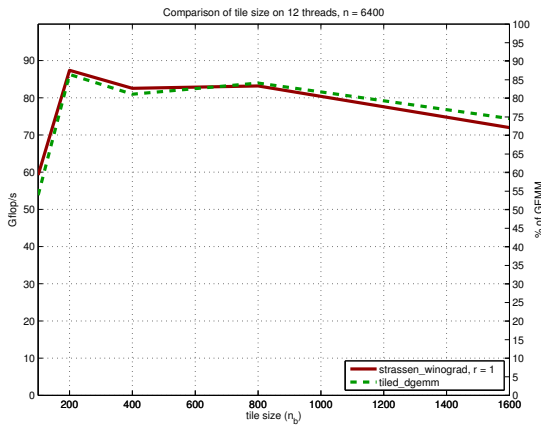
After this work was completed, Ballard et al. published work which places Strassen-Winograd in a parallel distributed environment [9]. The paper is well written and they do show improvement over the standard algorithm for matrices of order over 94,000. Their algorithm is communication optimal and applies better to the distributed environment than the shared memory environment seeing that we do not have as much control over the memory distribution.

6.4 Experimental results

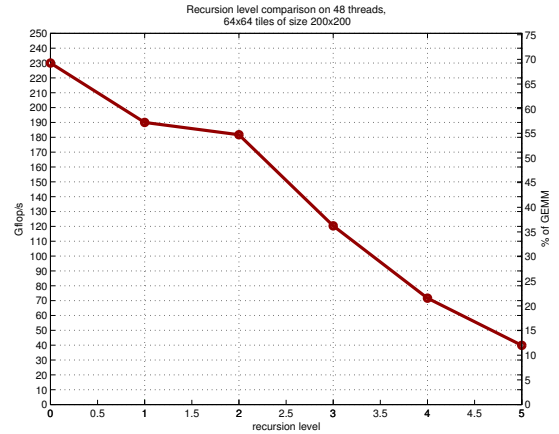
All experiments were performed on a 48-core machine composed of eight hexa-core AMD Opteron 8439 SE (codename Istanbul) processors running at 2.8 GHz.

Each core has a theoretical peak of 11.2 Gflop/s with a peak of 537.6 Gflop/s for the whole machine. The Istanbul micro-architecture is a NUMA architecture where each socket has 6 MB of level-3 cache and each processor has a 512 KB level-2 cache and a 128 KB level-1 cache. After having benchmarked the AMD ACML and Intel MKL BLAS libraries, we selected MKL (10.2) since it appeared to be slightly faster in our experimental context. Using MKL, for DGEMM each core has a peak of 9.7 Gflop/s with a peak of 465.6 Gflops/s for the whole machine. Linux 2.6.32 and Intel Compilers 11.1 were also used in conjunction with PLASMA 2.3.1.

The parameter for tile size has a direct effect on the amount of data movement and the efficiency of the kernels. Figure 6.4a presents the performance comparison for varying tile sizes indicating $n_b = 200$ as the most efficient. As expected, it is also evidenced that the efficiency of the Strassen-Winograd algorithm is dependent upon the efficiency of the tiled DGEMM. When running on 48 threads, increasing the recursion level decreases the performance of the algorithm as depicted in Figure 6.4 ($r = 0$ is the tiled matrix-matrix multiplication).



(a) Tile size comparison



(b) Recursion level comparison

Figure 6.4: Comparison of tuning parameters n_b and r .

Our implementation allows for the tuning of the recursion level and can range from one recursion up to full recursion. In Figure 6.4b, matrices of 64×64 tiles are

used and the recursion level ranges from one to five. Although $r_{min} = 2$ for 64×64 tiles, the best performance using 48 threads is seen at $r = 1$. This is due to the amount of parallelism lost by having the critical path length increase as the recursion level increases which offsets any gains of the reduction in tasks, and computational cost and complexity.

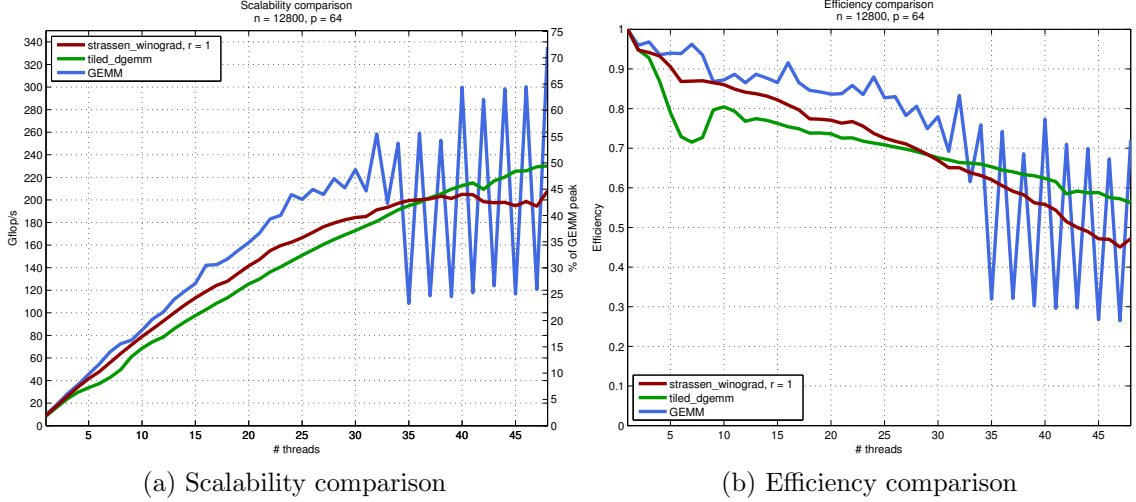


Figure 6.5: Scalability and Efficiency comparisons on 48 threads with matrices of 64×64 tiles and $n_b = 200$.

Figures 6.5a and 6.5b illustrate the performance and efficiency reached by the Strassen-Winograd implementation, with $r = 1$, as compared to the multithreaded MKL DGEMM and the tiled DGEMM implementation. The Strassen-Winograd implementation outperforms the tiled DGEMM up to the point where we lose parallelism. However, both show sub-par performance when compared to the multithreaded MKL implementation.

If we run on 12 cores (typical current architecture for a node) then we do outperform tiled DGEMM (Algorithm 6.1) if a loss of parallelism is not a factor and there is not too much data movement, i.e., keep n_b small enough so that more tiles fit into the cache but large enough to retain the efficiency of the GEMM kernel. Depicted in Figure 6.6, the performance of the Strassen-Winograd algorithm is best when $r = r_{min}$

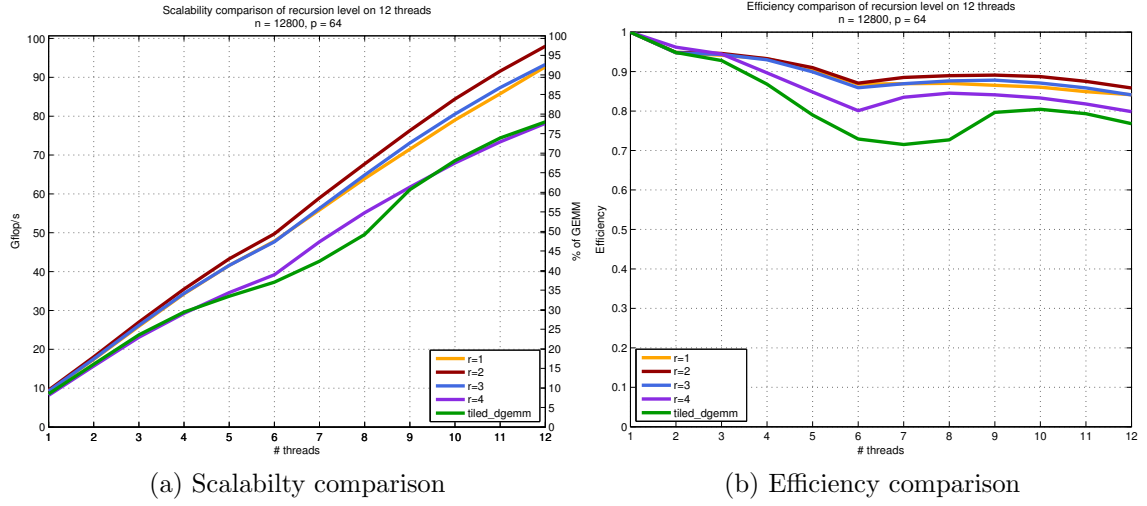


Figure 6.6: Scalability and efficiency comparisons executed on 12 threads with matrices of 64×64 tiles and $n_b = 200$.

since the number of tasks and computational complexity is minimized which reflects the analysis of § 6.2.

6.5 Conclusion

In this chapter we have shown and analyzed an implementation of the Strassen-Winograd algorithm in a tiled framework and provided comparisons to the multithreaded MKL standard library as well as a tiled matrix multiplication. The interest in this implementation is that it can support any level of recursion and any level of parallelism through the use of the recursion and tile size parameters.

Albeit that our implementation did not perform as well as the highly tuned and optimized multithreaded MKL library, on 12 cores with 2 levels of recursion, its performance was only lower by about 2%. Ultimately, it is a formidable task to surpass the MKL implementation considering the computational complexity of a recursive tiled algorithm.

7. Conclusion

In this thesis, we have studied the tiled algorithms both theoretically and experimentally. Our aim was to alleviate the constraints of memory boundedness, task granularity, and synchronicity imposed by the LAPACK library as brought to light in the Cholesky Decomposition example in the introduction. Moreover, we have also detailed that one may translate the LAPACK algorithms directly to tiled algorithms while in other cases a new approach may provide better performance gains.

In the study of the Cholesky Inversion, the tiled algorithms provided a unique insight into the interaction of the three distinct steps involved in the algorithm and how these may be intertwined. We had observed that the choice of the variant in the inversion of the triangular factor (Step 2) has a great impact on the performance of the algorithm. In the scope of unlimited number of processors, this choice can lead to an algorithm which performs the inversion of the matrix in almost the same amount of time it takes to do the Cholesky factorization. Moreover, we note that the combination of the variants with the shortest critical path length does not translate into the best performing pipelined algorithm. Even though variant 3 of Step 2 (the triangular inversion) did not provide us with the shortest critical path length within itself, combined with the other two steps, it does provide a Cholesky Inversion algorithm that performed the best overall.

We have also observed that a simple translation from the LAPACK routines may not provide a tiled algorithm with the best performance. We showed that loop-reversals are needed to alleviate anti-dependencies which negatively affect the performance of the tiled algorithms.

In the case where a tiled algorithm already existed, namely the QR Factorization, we made use of a new tiled algorithm to improve upon performance. These algorithms exhibit more parallelism than state-of-the-art implementations based on elimination trees. Using ideas from the 1970/80's, we have theoretically shown that the new

algorithm, GRASAP, is optimal in the scope of unbounded number of processors. We have provided accurate estimations for the length of the critical paths for all of these new tiled algorithms and have provided explicit formulas for some of the algorithms.

In the framework of bounded number of processors, our theoretical work has afforded a new bound which more accurately reflects the performance expectations of the tiled algorithms. In the case of the Cholesky Factorization, the schedule produced using the Critical Path Method proved to be within seven percent of the ALAP Derived bound indicating that this scheduling strategy is well suited for this application. The ALAP Derived bound has also been used as a tool to show that the optimality in the scope unbounded number of processors does not translate to optimality in the scheduling on a bounded number of processors.

Overall, the theoretical and experimental portions of this thesis give credence to the impact of tiled algorithms for multi/many-core architectures.

APPENDIX A. Integer Programming Formulation of Tiled QR

A.1 IP Formulation

We formulate the problem in question using integer programming. An equivalent binary programming formulation was also constructed (with time as an additional index), but we found the integer formulations were more quickly solved.

Let i, j , and h denote rows (ranging $1, \dots, p$); k, l, l_1 denote columns (ranging $1, \dots, q$); r, s , and t denote time steps (ranging $1, \dots, T$). The upper bound on the number of time steps, (T) , may come from any existing algorithm (greedy, ASAP, GRASP, etc.). Let the decision variables be defined as follows:

A.1.1 Variables

Let all t be an integer ranging from zero to T denoting the time we complete the following tasks (and $t = 0$ means the task is never performed).

- **UNMQR:** Complete applying the reflectors from GEQRT across the row. (Requires 6 units of time.)

$$w_{ikl} = t \in [0, T] : \text{ if we finish the update of tile } (i, k) \text{ at time } t. \quad (\text{A.1})$$

(This update was necessitated by $x_{il} = s : l < k, s < t$).

- **GEQRT:** Factor a square tile into a triangle. (Requires 4 units of time.)

$$x_{ik} = t \in [0, T] : \text{ if we complete triangularization of tile } (i, k) \text{ at} \quad (\text{A.2})$$

the end of time unit t .

- **TTMQR:** Update the entries in two rows after TTQRT. (Requires 6 units of time.)

$$y_{ijkl} = t \in [0, T] : \text{ if we finish the update of tile } (i, k) \text{ and } (j, k) \text{ at} \quad (\text{A.3})$$

the end of time unit t .

(This update was necessitated by $z_{ijl} = s : l < k, s < t$)

- **TTQRT:** Cancel one triangular tile using another triangle tile. (Requires 2 units of time.)

$$z_{ijk} = t \in [0, T] : \quad \text{if we complete zeroing tile } (i, k) \text{ using tile } (j, k) \quad (\text{A.4})$$

at the end of time unit t .

For the y and z actions, it is useful to have a binary variable which is 1 if the action occurs. Explicitly,

$$\hat{y}_{ijkl} = \begin{cases} 1 : & \text{if } y_{ijkl} > 0 \\ 0 : & \text{otherwise} \end{cases} \quad \hat{z}_{ijk} = \begin{cases} 1 : & \text{if } z_{ijk} > 0 \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.5})$$

A.1.2 Constraints

1. Time constraints for each of the four actions:

(a) Time for w_{ikl}

- i. w_{ikl} must occur at least 3 time steps after earlier w updates.

$$w_{ikl} \geq w_{ikl_1} + 3 \quad \forall k \geq 2, i \geq l, l_1 < l < k \quad (\text{A.6})$$

- ii. w_{ikl} must occur at least 3 time steps after earlier y updates.

$$w_{ikl} \geq y_{ijkl_1} + y_{jikl_1} + 3 \quad \forall k \geq 2, i \geq l, j, l_1 < l < k \quad (\text{A.7})$$

- iii. w_{ikl} must occur at least 3 time steps before y updates in the current column.

$$w_{ikl} + 3 \leq y_{ijkl} + y_{jikl} + (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T \quad \forall i, j, l < k, k \geq 2 \quad (\text{A.8})$$

- iv. w_{ikl} must occur at least 2 time steps before x .

$$w_{ikl} + 2 \leq x_{ik} \quad \forall i \geq k \geq 2, l < k \quad (\text{A.9})$$

- v. w_{ikl} must occur at least 1 time step before z actions.

$$w_{ikl} + 1 \leq z_{ijk} + z_{jik} + (1 - \hat{z}_{ijk} - \hat{z}_{jik})t \quad \forall i, j, k \geq 2, l < k \quad (\text{A.10})$$

(b) Time for x_{ik}

i. x_{ik} must occur 2 time steps after any w updates. (Identical to equation (A.9).)

ii. x_{ik} must occur 2 time steps after any y updates.

$$x_{ik} \geq y_{ijkl} + y_{jikl} + 2 \quad \forall j, l, i \geq k, l < k \quad (\text{A.11})$$

iii. x_{ik} must occur 1 time steps before any z action.

$$x_{ik} + 1 \leq z_{ijk} + z_{jik} + (1 - \hat{z}_{ijk} - \hat{z}_{jik})T \quad \forall i \geq k, j \quad (\text{A.12})$$

(c) Time for y_{ijkl}

i. y_{ijkl} must occur 3 time steps after w updates originating from the same column. (Identical to equation A.8.)

ii. y_{ijkl} must occur 2 time steps before any x_{ik} or x_{jk} . (Identical to equation A.11.)

iii. y_{ijkl} must be 3 time steps before or after any y action involving rows i or j .

We must enforce

$$y_{ijkl} + y_{jikl} + 3 \leq y_{hikl} + y_{ihkl} + (1 - \hat{y}_{hikl} - \hat{y}_{ihkl})T$$

or

$$y_{hikl} + y_{ihkl} + 3 \leq y_{ijkl} + y_{jikl} + (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T$$

And

$$y_{ijkl} + y_{jikl} + 3 \leq y_{hjkl} + y_{jhkl} + (1 - \hat{y}_{hjkl} - \hat{y}_{jhkl})T$$

or

$$y_{hjkl} + y_{jhkl} + 3 \leq y_{ijkl} + y_{jikl} + (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T$$

We define the binary variables δ_{hijkl}^1 , δ_{hijkl}^2 , δ_{hijkl}^3 , and δ_{hijkl}^4 , and include the disjunctive constraints

$$y_{ijkl} + y_{jikl} + 3 \leq y_{hikl} + y_{ihkl} + (1 - \hat{y}_{hikl} - \hat{y}_{ihkl})T + \delta_{hijkl}^1 T \quad (\text{A.13})$$

$$\forall h, i, j, l < k \geq 2$$

$$y_{hikl} + y_{ihkl} + 3 \leq y_{ijkl} + y_{jikl} + (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T + \delta_{hijkl}^2 T \quad (\text{A.14})$$

$$\forall h, i, j, l < k \geq 2$$

$$\delta_{hijkl}^1 + \delta_{hijkl}^2 \geq 1 \quad \forall h, i, j, l < k \geq 2 \quad (\text{A.15})$$

and

$$y_{ijkl} + y_{jikl} + 3 \leq y_{hjkl} + y_{jhkl} + (1 - \hat{y}_{hjkl} - \hat{y}_{jhkl})T + \delta_{hijkl}^3 T \quad (\text{A.16})$$

$$\forall h, i, j, l < k \geq 2$$

$$y_{hjkl} + y_{jhkl} + 3 \leq y_{ijkl} + y_{jikl} + (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T + \delta_{hijkl}^4 T \quad (\text{A.17})$$

$$\forall h, i, j, l < k \geq 2$$

$$\delta_{hijkl}^3 + \delta_{hijkl}^4 \geq 1 \quad \forall h, i, j, l < k \geq 2 \quad (\text{A.18})$$

iv. y_{ijkl} must occur 3 time steps before any z action involving rows i or j .

$$y_{ijkl} + 3 \leq z_{hik} + z_{ihk} + (1 - \hat{z}_{hik} - \hat{z}_{ihk})T \quad \forall h, i, j, k > l, k \geq 2$$

$$y_{ijkl} + 3 \leq z_{hjk} + z_{jhk} + (1 - \hat{z}_{hjk} - \hat{z}_{jhk})T \quad \forall h, i, j, k > l, k \geq 2 \quad (\text{A.19})$$

(d) Time for z_{ijk}

i. z_{ijk} must occur 1 time step after any w action. (Identical to equation A.10.)

ii. z_{ijk} must occur 1 time step after any y action. (Identical to equation A.19.)

iii. z_{ijk} must occur 1 time step after any x action. (Identical to equation A.12.)

iv. z_{ijk} must occur 1 time step before or after any other z action.

A. Case 1. We use (i, k) to zero (j, k) and (i, k) to zero (h, k)

We need to enforce

$$z_{jik} + 1 \leq z_{hik} + (1 - \hat{z}_{hik})T$$

or

$$z_{hik} + 1 \leq z_{jik} + (1 - \hat{z}_{jik})T$$

To do this, we define binary variables δ_{hijk}^5 and δ_{hijk}^6 and include the disjunctive constraints as follows.

$$z_{jik} + 1 \leq z_{hik} + (1 - \hat{z}_{hik})T + \delta_{hijk}^5 T \quad \forall h, i, j, k \quad (\text{A.20})$$

$$z_{hik} + 1 \leq z_{jik} + (1 - \hat{z}_{jik})T + \delta_{hijk}^6 T \quad \forall h, i, j, k \quad (\text{A.21})$$

$$\delta_{hijk}^5 + \delta_{hijk}^6 \geq 1 \quad \forall h, i, j, k \quad (\text{A.22})$$

B. Case 2. We use (i, k) to zero (j, k) and (h, k) to zero (i, k) We need to enforce

$$z_{jik} + 1 \leq z_{ihk} + (1 - \hat{z}_{ihk})T \quad \forall h, i, j, k (i > j??) \quad (\text{A.23})$$

2. A tile can't zero itself, (and hence we can't update just a single row afterwards).

$$z_{iik} = 0 \quad \forall i, k \quad y_{iikl} = 0 \quad \forall i, k, l \quad (\text{A.24})$$

3. Both tiles involved in TTQRT must be triangles before one can zero another

$$x_{ik} \leq (1 - \hat{z}_{ijk})T + z_{ijk} \quad \forall i, j, k, \quad x_{jk} \leq (1 - \hat{z}_{ijk})T + z_{ijk} \quad \forall i, j, k \quad (\text{A.25})$$

4. Force updates after triangle and zeroing actions.

(a) After a tile is triangularized, updates must occur in the next column.

$$x_{ik} \leq w_{ilk} - 3 \quad \forall i, k < q, i \geq k, l > k \quad (\text{A.26})$$

(b) After a tile is zeroed, updates must occur in the next column.

$$z_{ijk} \leq (y_{ijlk} + y_{jilk}) \quad \forall i, j, k < l \quad (\text{A.27})$$

5. The updates of (i, k) (arising from pivot l) from triangularizing must occur before updates from zeroing involving (i, k) (also arising from pivot l).

$$w_{ikl} \leq (1 - \hat{y}_{ijkl} - \hat{y}_{jikl})T + y_{ijlk} + y_{jilk} \quad \forall i, j, k > l \quad (\text{A.28})$$

6. No updates to a tile can occur after triangularization.

$$x_{ik} \geq w_{ikl} \quad \forall i \geq k, k > l \quad x_{ik} \geq y_{ijkl} + y_{jikl} \quad \forall i \geq k, j, k > l \quad (\text{A.29})$$

7. After a tile (i, k) is zeroed, we can't use it to zero.

$$z_{ijk} \geq z_{hik} \quad \forall h, i, j, k \quad (\text{A.30})$$

8. Tiles on or below the diagonal must be triangularized at some point, (and we can't finish a triangularization until time step 2.)

$$x_{ik} \geq 2 \quad \forall i \geq k \quad (\text{A.31})$$

9. Tiles strictly below the diagonal must be zeroed at some point.

$$\sum_j \hat{z}_{ijk} = 1 \quad \forall i > k \quad (\text{A.32})$$

10. Can't triangularize above the diagonal.

$$x_{ik} = 0 \quad \forall i < k \quad (\text{A.33})$$

11. Force binary variables

$$\hat{y}_{ijkl} \leq y_{ijkl} \quad \hat{y}_{ijkl} * T \geq y_{ijkl} \quad \forall i, j, k, l \quad (\text{A.34})$$

$$\hat{z}_{ijk} \leq z_{ijk} \quad \hat{z}_{ijk} * T \geq z_{ijk} \quad \forall i, j, k \quad (\text{A.35})$$

A.1.2.1 Precedence constraints

The most cumbersome constraints to formulate are those forcing the corresponding TTQRT and TTMQR operations to be executed in the same order.

For each tile (i, k) involved in a zeroing process with (j, k) and (h, k) , the order of the updates must follow the order of the zeroing processes. We want $z_{jik} < z_{ihk}$ (for some h) or $z_{jik} < z_{hik}$ (for some h), to force $y_{ij(k+1)k} < y_{ih(k+1)k}$.

1. a^1

$$a_{hijk}^1 = \begin{cases} 1 : & \text{if we [use } (i, k) \text{ to zero } (h, k)] \text{ and also [use } (i, k) \\ & \text{to zero } (j, k)] \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.36})$$

$$a_{hijk}^1 \leq \hat{z}_{hik}$$

$$a_{hijk}^1 \leq \hat{z}_{jik} \quad (\text{A.37})$$

$$a_{hijk}^1 + 1 \geq \hat{z}_{hik} + \hat{z}_{jik}$$

2. a^2

$$a_{hijk}^2 = \begin{cases} 1 : & \text{if we [use } (h, k) \text{ to zero } (i, k)] \text{ after [using } (i, k) \text{ to} \\ & \text{zero } (j, k)] \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.38})$$

$$a_{hijk}^2 \leq \hat{z}_{ihk}$$

$$a_{hijk}^2 \leq \hat{z}_{jik} \quad (\text{A.39})$$

$$a_{hijk}^2 + 1 \geq \hat{z}_{ihk} + \hat{z}_{jik}$$

3. b

$$b_{hijk} = \begin{cases} 1 : & \text{if } [z_{hik} > z_{jik}] \text{ regardless of whether } [z_{hik} = 0] \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.40})$$

$$\begin{aligned}
Tb_{hijk} &\geq z_{hik} - z_{jik} \\
T(b_{hijk} - 1) &\leq z_{hik} - z_{jik}
\end{aligned} \tag{A.41}$$

4. c^1

$$c_{hijk}^1 = \begin{cases} 1 : & \text{if } [z_{hik} > z_{jik}] \\ 0 : & \text{otherwise} \end{cases} \tag{A.42}$$

$$\begin{aligned}
c_{hijk}^1 &\leq a_{hijk}^1 \\
c_{hijk}^1 &\leq b_{hijk} \\
c_{hijk}^1 + 1 &\geq a_{hijk}^1 + b_{hijk}
\end{aligned} \tag{A.43}$$

5. c

$$c_{hijk} = \begin{cases} 1 : & \text{if zeroing actions in column } k \text{ of rows } h \text{ and } i \text{ to} \\ & \text{occur after the zeroing actions of rows } i \text{ and } j \\ 0 : & \text{otherwise} \end{cases} \tag{A.44}$$

$$\begin{aligned}
c_{hijk} &\geq c_{hijk}^1 \\
c_{hijk} &\geq a_{hijk}^2 \\
c_{hijk} &\leq c_{hijk}^1 + a_{hijk}^2
\end{aligned} \tag{A.45}$$

So we now have a variable c_{hijk} that is 1 when updates of row h and i must come before the updates of i and j . We can define similar variables for the updates rather than the zeros.

6. d

$$d_{hijk} = \begin{cases} 1 : & \text{if we [update } (i, k) \text{ and } (h, k) \text{ together] and also} \\ & \text{[update } (i, k) \text{ and } (j, k) \text{ together]. (All updates} \\ & \text{occur because of zeroing in column } k - 1) \\ 0 : & \text{otherwise} \end{cases} \tag{A.46}$$

$$\begin{aligned}
d_{hijk} &\leq \hat{y}_{hik(k-1)} + \hat{y}_{ihk(k-1)} \\
d_{hijk} &\leq \hat{y}_{jik(k-1)} + \hat{y}_{ijk(k-1)} \\
d_{hijk} + 1 &\geq \hat{y}_{hik(k-1)} + \hat{y}_{ihk(k-1)} + \hat{y}_{jik(k-1)} + \hat{y}_{ijk(k-1)}
\end{aligned} \tag{A.47}$$

7. e

$$e_{hijk} = \begin{cases} 1 : & \text{if updates in column } k \text{ of rows } h \text{ and } i \text{ to occur} \\ & \text{after updates of } i \text{ and } j \text{ or the updates in } i \text{ and } j \\ & \text{never happen} \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.48})$$

$$\begin{aligned} Te_{hijk} &\geq (y_{hik(k-1)} + y_{ihk(k-1)}) - (y_{jik(k-1)} + y_{ijk(k-1)}) \\ T(e_{hijk} - 1) &\leq (y_{hik(k-1)} + y_{ihk(k-1)}) - (y_{jik(k-1)} + y_{ijk(k-1)}) \end{aligned} \quad (\text{A.49})$$

8. f

$$f_{hijk} = \begin{cases} 1 : & \text{if the updates in column } k \text{ of rows } h \text{ and } i \text{ to occur} \\ & \text{after updates of } i \text{ and } j \\ 0 : & \text{otherwise} \end{cases} \quad (\text{A.50})$$

$$\begin{aligned} f_{hijk} &\geq d_{hijk} \\ f_{hijk} &\geq e_{hijk} \\ f_{hijk} &\leq d_{hijk} + e_{hijk} \end{aligned} \quad (\text{A.51})$$

Lastly, to force the updates in order:

$$c_{hijk} \leq f_{hijl} \quad \forall h, i, j, k < l \quad (\text{A.52})$$

A.1.3 Objective function

Let `total_time` be a variable such that

$$\begin{aligned} total_time &\geq w_{ikl} && \forall i, k, l \\ total_time &\geq x_{ik} && \forall i, k \\ total_time &\geq y_{ijk} && \forall i, j, k \\ total_time &\geq z_{ijk} && \forall i, j, k \end{aligned}$$

Then our objective function is:

$$\min total_time \quad (\text{A.53})$$

REFERENCES

- [1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932–943, may 2011.
- [2] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, and H. Ltaief. PLASMA Users’ Guide. Technical report, ICL, UTK, 2009.
- [3] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC’09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.
- [4] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *Proceedings of the 9th international conference on High performance computing for computational science, VECPAR’10*, pages 129–138, Berlin, Heidelberg, 2011. Springer-Verlag.
- [5] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC ’09)*, pages 1–12. IEEE Computer Society Press, 2009.
- [6] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.
- [7] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. SIAM, 1992.
- [8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiawicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.
- [9] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for Strassen’s matrix multiplication. In *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures, SPAA ’12*, pages 193–204, New York, NY, USA, 2012. ACM.

- [10] P. Bientinesi, B. Gunter, and R. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.
- [11] Bilel Hadri and Hatem Ltaief and Emmanuel Agullo and Jack Dongarra. Enhancing Parallelism of Tile QR Factorization for Multicore Architectures. Technical Report 222, LAPACK Working Note, 2009.
- [12] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234–235, 1979.
- [13] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users’ Guide*. SIAM, 1997.
- [14] Susan Blackford and Jack J. Dongarra. Installation Guide for LAPACK. Technical Report 41, LAPACK Working Note, jun 1999. originally released March 1992.
- [15] BLAS. Basic Linear Algebra Subprograms. <http://www.netlib.org/blas/>.
- [16] Henricus Bouwmeester and Julien Langou. A Critical Path Approach to Analyzing Parallelism of Algorithmic Variants. Application to Cholesky Inversion. *CoRR*, abs/1010.2000, 2010.
- [17] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd’s matrix multiplication algorithm. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC ’09, pages 55–62, New York, NY, USA, 2009. ACM.
- [18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008.
- [19] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
- [20] H. Casanova, A. Legrand, and Y. Robert. *Parallel algorithms*. Chapman & Hall/CRC numerical analysis and scientific computing. CRC Press, 2009.
- [21] Chung chiang Chou, Yuefan Deng, Gang Li, and Yuan Wang. Parallelizing Strassen’s Method for Matrix Multiplication on Distributed-Memory MIMD Architectures. In *Computers for Mathematics with Applications*, 1994.
- [22] N. Christofides. *Graph Theory: An algorithmic Approach*. Academic Press, 1975.

- [23] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM.
- [24] Michel Cosnard, Jean-Michel Muller, and Yves Robert. Parallel QR decomposition of a rectangular matrix. *Numerische Mathematik*, 48:239–249, 1986.
- [25] Michel Cosnard and Yves Robert. Complexity of parallel QR factorization. *Journal of the A.C.M.*, 33(4):712–723, 1986.
- [26] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice. Technical Report 204, LAPACK Working Note, 2008.
- [27] Craig C. Douglas, Michael Heroux, Gordon Slishman, Roger M. Smith, and Roger M. Gemm. A Portable Level 3 Blas Winograd Variant Of Strassen's Matrix-Matrix Multiply Algorithm, 1994.
- [28] R. Eigenmann, J. Hoefflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks®. *IEEE Trans. Parallel Distrib. Syst.*, 9(1):5–23, 1998.
- [29] Emmanuel Agullo and Jack Dongarra and Rajib Nath and Stanimire Tomov. A Fully Empirical Autotuned Dense QR Factorization For Multicore Architectures. Technical Report 242, LAPACK Working Note, 2011.
- [30] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. In *24th IEEE Int. Parallel Distributed Processing Symposium IPDPS'10*, 2010.
- [31] Henricus Bouwmeester and Mathias Jacquelin and Julien Langou and Yves Robert. Tiled QR factorization algorithms. Technical Report 7601, INRIA, France, apr 2011. Available at <http://hal.inria.fr/docs/00/58/62/39/PDF/RR-7601.pdf>.
- [32] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
- [33] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. *University of Tennessee CS Tech. Report, UT-CS-09-643*, 2009.
- [34] J. Kurzak and J. Dongarra. QR factorization for the Cell Broadband Engine. *Sci. Program.*, 17(1-2):31–42, 2009.
- [35] R. E. Lord, J. S. Kowalik, and S. P. Kumar. Solving Linear Algebraic Equations on an MIMD Computer. *J. ACM*, 30(1):103–117, January 1983.

- [36] J.J. Modi and M.R.B. Clarke. An alternative Givens ordering. *Numerische Mathematik*, 43:83–90, 1984.
- [37] Mounir Marrakchi and Yves Robert. Optimal algorithms for Gaussian elimination on an MIMD computer. *Parallel Computing*, 12(2):183 – 194, 1989.
- [38] Yuji Nakatsukasa and Nicholas J. Higham. Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD. <http://eprints.ma.man.ac.uk/1824/>, preprint.
- [39] V. Ya. Pan. Strassen’s algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 166 –176, oct. 1978.
- [40] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007.
- [41] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.
- [42] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 6:28–38, 1993.
- [43] Yves Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Manchester University Press, 1991.
- [44] F. Romani. Some Properties of Disjoint Sums of Tensors Related to Matrix Multiplication. *SIAM Journal on Computing*, 11(2):263–267, 1982.
- [45] A.H. Sameh and D.J. Kuck. On stable parallel linear systems solvers. *J. ACM*, 25:81–91, 1978.
- [46] A. Schönhage. Partial and Total Matrix Multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.
- [47] SimGrid. URL: <http://simgrid.gforge.inria.fr>.
- [48] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.
- [49] Frédéric Suter. Mixed Parallel Implementations of the Top Level Step of Strassen and Winograd Matrix Multiplication Algorithms. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS’01) - Volume 1*, IPDPS ’01, pages 10008.2–, Washington, DC, USA, 2001. IEEE Computer Society.

- [50] H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [51] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38:1621–1642, December 2008.
- [52] Christopher G. Willard and Addison Snell Laura Segervall. HPC User Site Census: Systems. <http://www.intersect360.com/industry/reports.php?id=67>, 2012.
- [53] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009.
- [54] Shmuel Winograd. On Multiplication of 2×2 Matrices. *Linear Algebra and Its Applications*, 4:381–388, 1971.